US006453358B1

(12) United States Patent
Michels et al.

(10) **Patent No.: US 6,453,358 B1**
(45) **Date of Patent: Sep. 17, 2002**

(54) **NETWORK SWITCHING DEVICE WITH CONCURRENT KEY LOOKUPS**

(75) Inventors: **Timothy Scott Michels**, Spokane; **James E. Cathey**, Greenacres; **Greg W. Davis**; **Bernard N. Daines**, both of Spokane, all of WA (US)

(73) Assignee: **Alcatel Internetworking (PE), Inc.**, Spokane, WA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 23 days.

(21) Appl. No.: **09/776,940**

(22) Filed: **Sep. 6, 2000**

Related U.S. Application Data

(63) Continuation of application No. 09/166,707, filed on Oct. 5, 1998, now Pat. No. 6,161,144.
(60) Provisional application No. 60/072,280, filed on Jan. 23, 1998.

(51) **Int. Cl.**^{7} **G06F 13/00**
(52) **U.S. Cl.** **709/238**; 711/157; 370/392; 712/300
(58) **Field of Search** 709/230, 238, 709/242, 245, 249, 250; 370/389, 392; 711/149, 150, 157, 168, 169; 712/300

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,214,639	A		5/1993	Herion	370/60
5,386,413	A		1/1995	McAuley et al.	370/54
5,414,704	A		5/1995	Spinney	
5,459,724	A		10/1995	Jeffrey et al.	370/60
5,608,726	A		3/1997	Virgile	370/401
5,748,905	A		5/1998	Hauser et al.	
5,796,944	A	*	8/1998	Hill et al.	709/250
5,895,500	A		4/1999	Thomason et al.	
5,905,725	A		5/1999	Sindhu et al.	370/389
5,909,440	A		6/1999	Ferguson	
5,909,686	A		6/1999	Muller et al.	707/104
5,938,736	A		8/1999	Muller et al.	709/243
5,946,679	A	*	8/1999	Ahuja et al.	707/3
6,006,306	A		12/1999	Haywood et al.	711/108
6,011,795	A	*	1/2000	Varghese et al.	370/392
6,023,466	A		2/2000	Luijten et al.	
6,032,190	A	*	2/2000	Bremer et al.	709/238
6,161,144	A		12/2000	Michels	709/238
6,259,699	B1		7/2001	Opalka et al.	370/398

OTHER PUBLICATIONS

Johnson et al, U.S. application Ser. No. 09/166,343, filed Oct. 5, 1998.
Michels et al, U.S. application Ser. No. 09/166,603, filed Oct. 5, 1998.
Michels et al, U.S. application Ser. No. 09/166,620, filed Oct. 5, 1998.

* cited by examiner

Primary Examiner—Viet D. Vu
(74) *Attorney, Agent, or Firm*—Christie, Parker & Hale, LLP

(57) **ABSTRACT**

A switching device for forwarding network traffic to a desired destination on a network, such as a telephone or computer network. The switching device includes multiple ports and uses a lookup table to determine which port to forward network traffic over. The lookup table includes network addresses that are maintained in ascending or descending order. The switching device includes multiple binary search engines coupled in series including one or more precursor binary search engines and a final stage binary search engine. Together, the binary search engines perform an N iteration binary search. Additionally, a single search engine can perform multiple concurrent searches so that source and destination addresses can be obtained simultaneously and without wasted memory cycles.

14 Claims, 10 Drawing Sheets

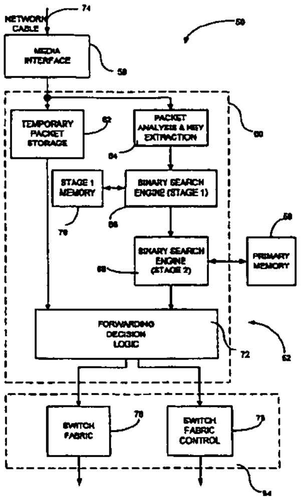


FIG. 1A
(PRIOR ART)

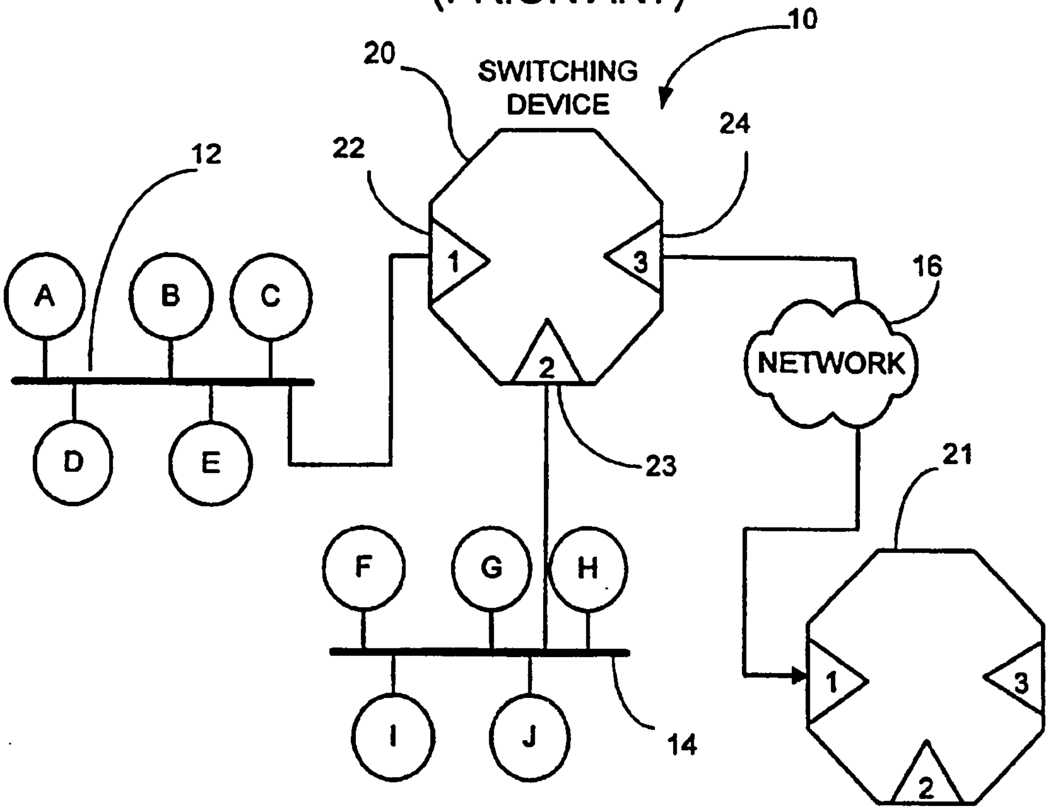


FIG. 1B
(PRIOR ART)

26

NETWORK ADDRESS	PORTS
A	1
M	3
N	3
H	2
F	2
J	2

28

30

FIG. 2

(PRIOR ART)

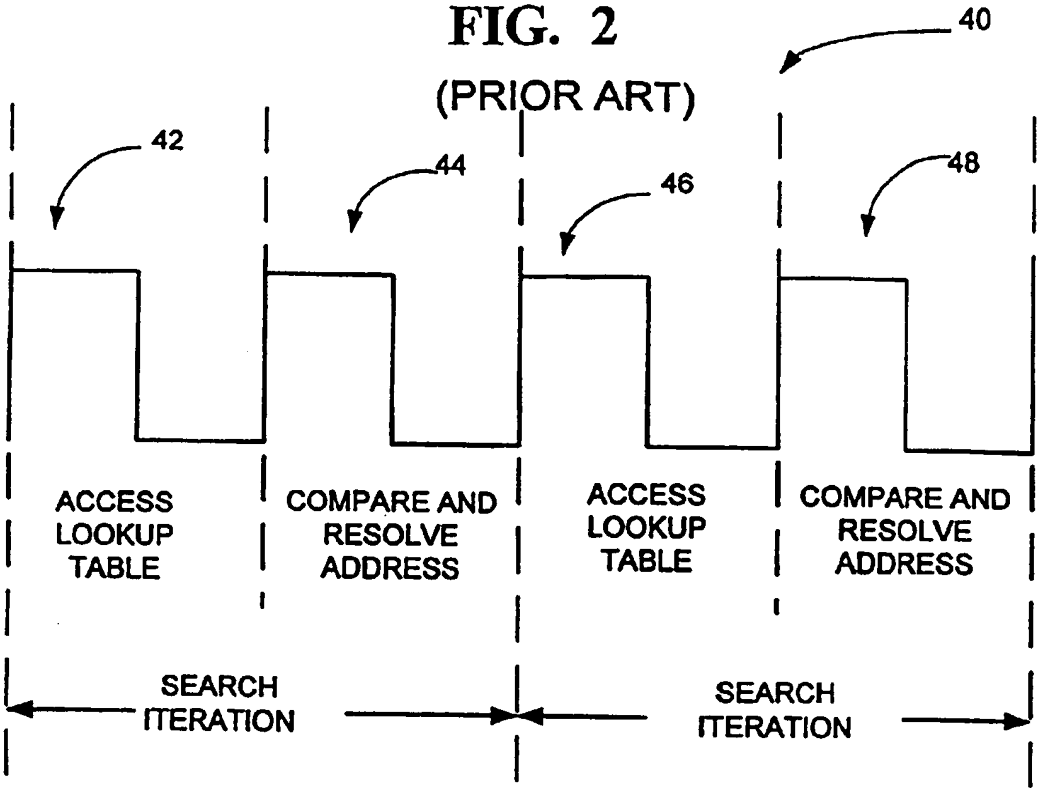


FIG. 3

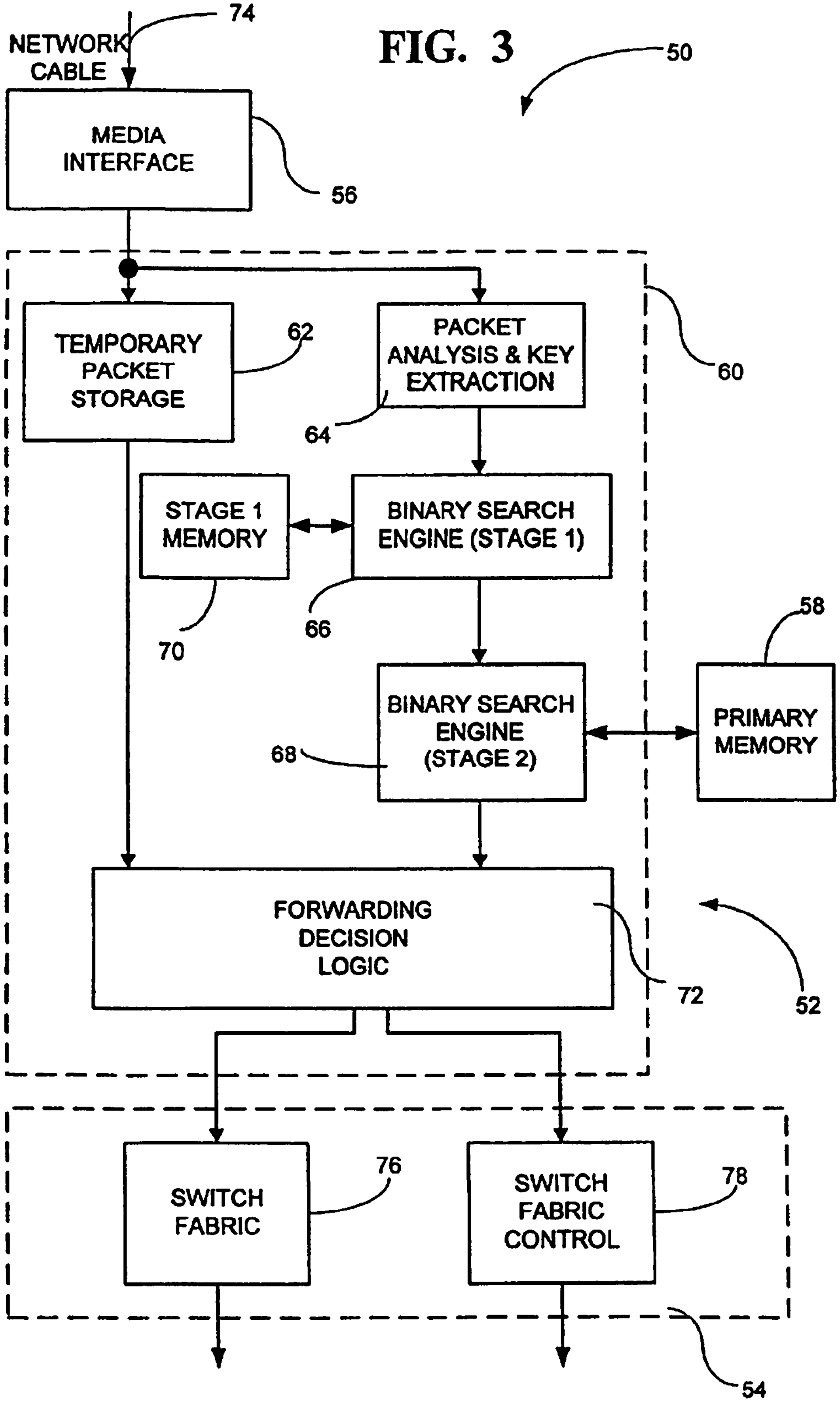
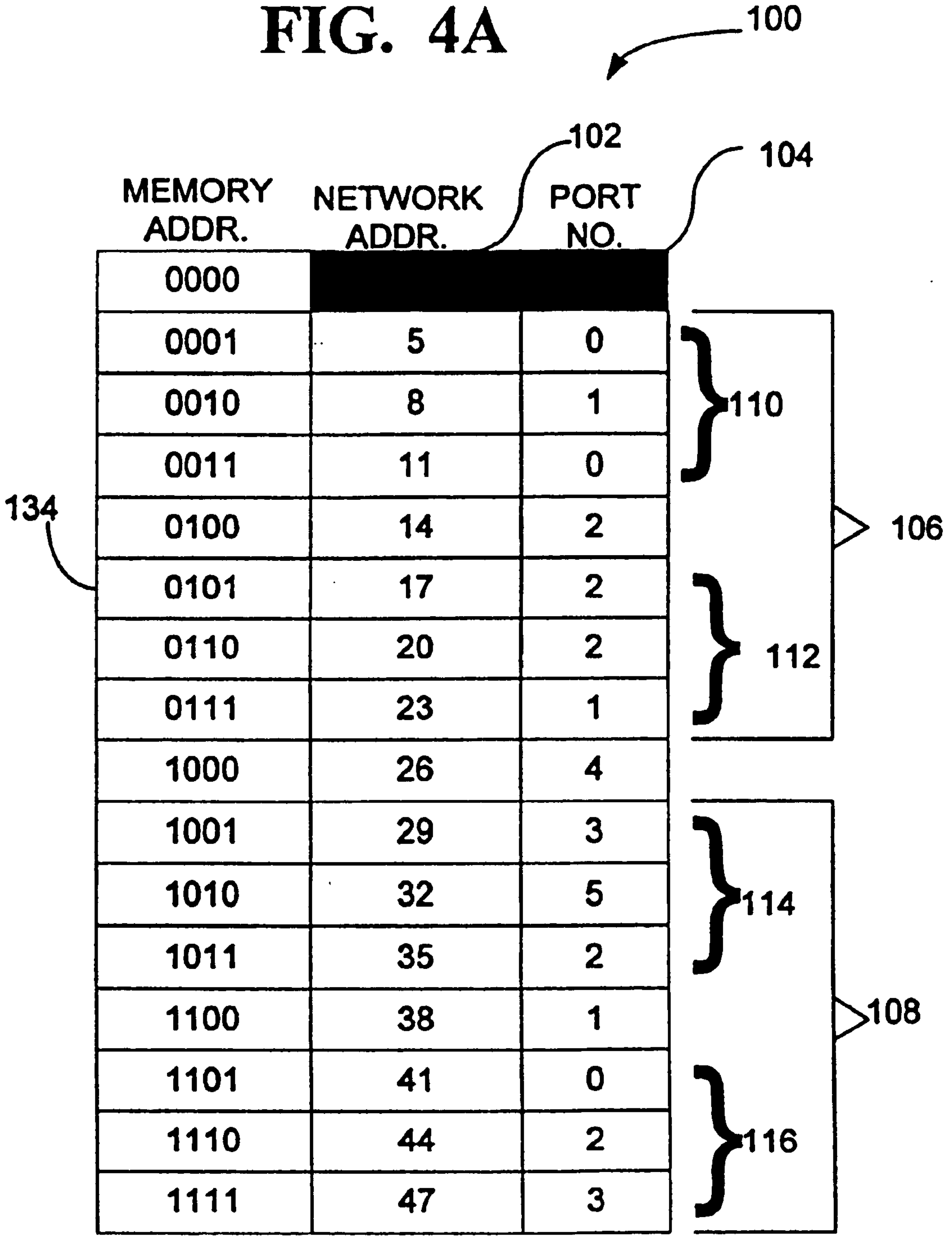

74
NETWORK CABLE
50
MEDIA INTERFACE
56
TEMPORARY PACKET STORAGE
62
PACKET ANALYSIS & KEY EXTRACTION
64
60
STAGE 1 MEMORY
BINARY SEARCH ENGINE (STAGE 1)
70
66
58
BINARY SEARCH ENGINE (STAGE 2)
68
PRIMARY MEMORY
FORWARDING DECISION LOGIC
72
52
SWITCH FABRIC
76
SWITCH FABRIC CONTROL
78
54

FIG. 4A



MEMORY ADDR.	NETWORK ADDR.	PORT NO.
0000		
0001	5	0
0010	8	1
0011	11	0
0100	14	2
0101	17	2
0110	20	2
0111	23	1
1000	26	4
1001	29	3
1010	32	5
1011	35	2
1100	38	1
1101	41	0
1110	44	2
1111	47	3

FIG. 4B

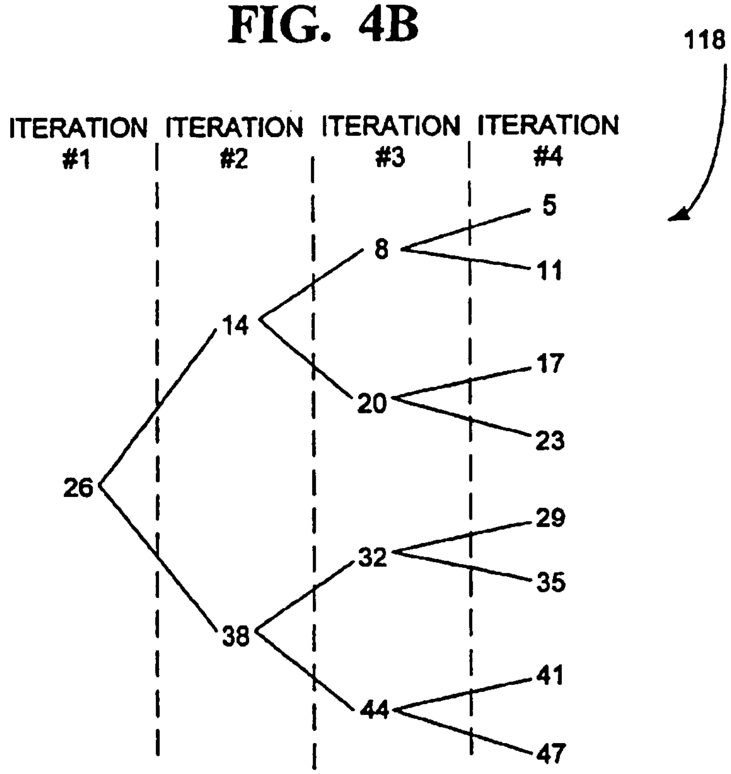


FIG. 4C

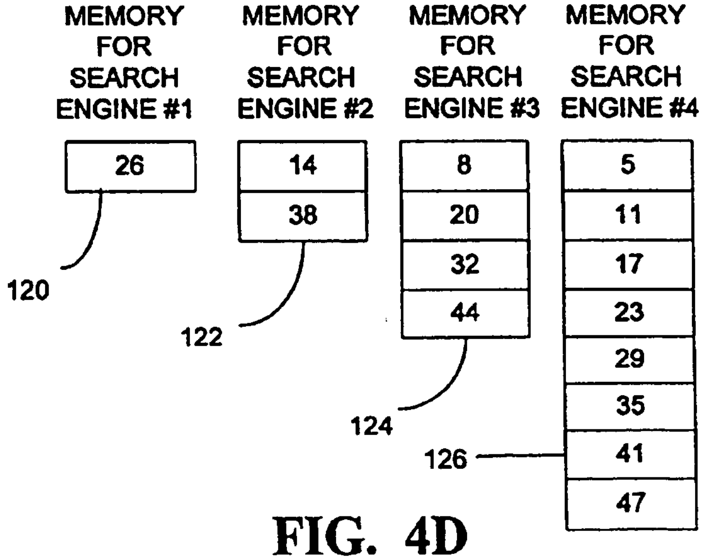

MEMORY FOR SEARCH ENGINE #1
MEMORY FOR SEARCH ENGINE #2
MEMORY FOR SEARCH ENGINE #3
MEMORY FOR SEARCH ENGINE #4
26
14
38
8
20
32
44
5
11
17
23
29
35
41
47
120
122
124
126

FIG. 4D

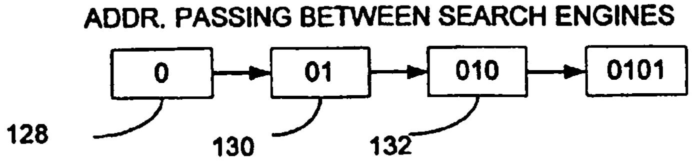


FIG. 5

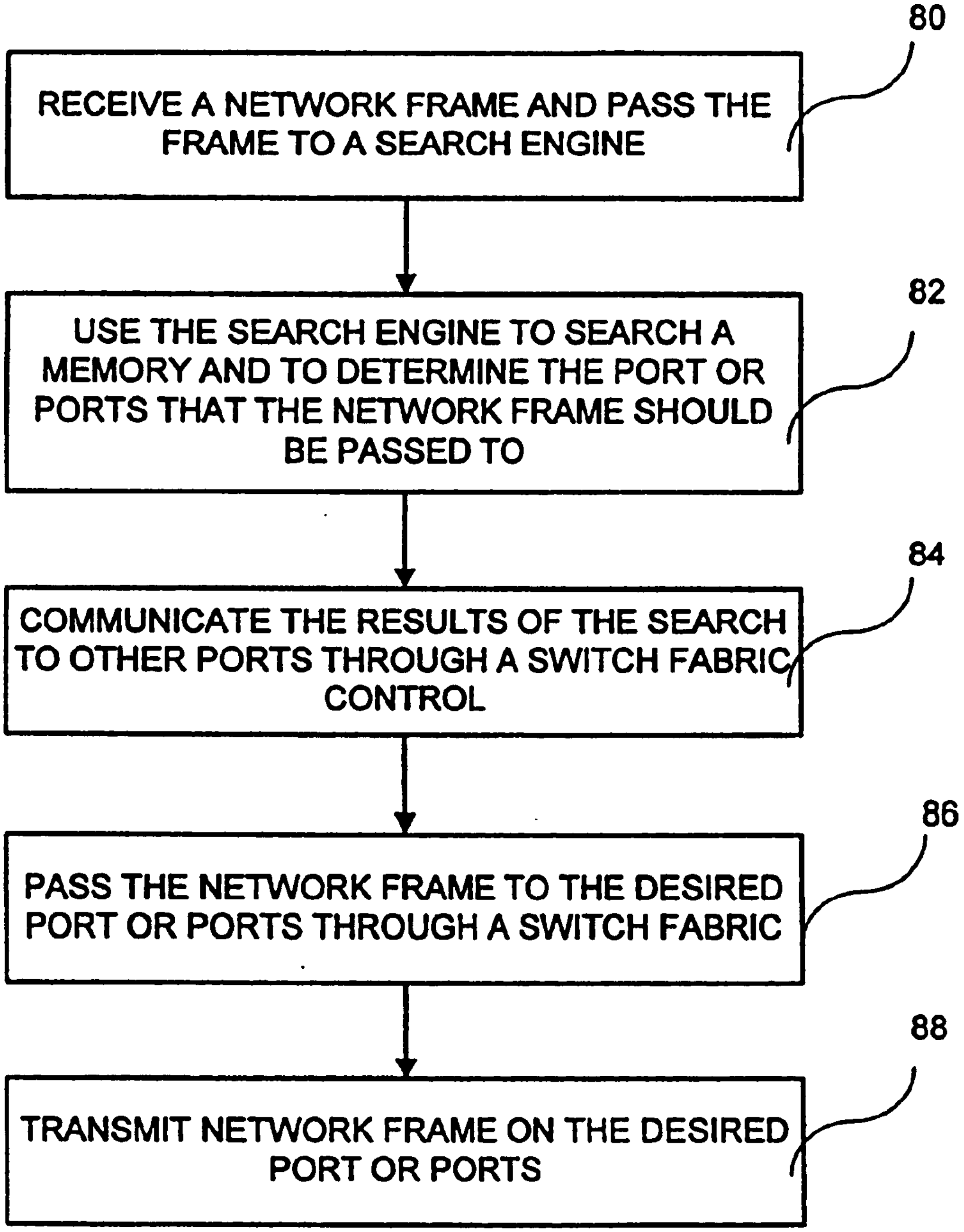

80
RECEIVE A NETWORK FRAME AND PASS THE FRAME TO A SEARCH ENGINE
82
USE THE SEARCH ENGINE TO SEARCH A MEMORY AND TO DETERMINE THE PORT OR PORTS THAT THE NETWORK FRAME SHOULD BE PASSED TO
84
COMMUNICATE THE RESULTS OF THE SEARCH TO OTHER PORTS THROUGH A SWITCH FABRIC CONTROL
86
PASS THE NETWORK FRAME TO THE DESIRED PORT OR PORTS THROUGH A SWITCH FABRIC
88
TRANSMIT NETWORK FRAME ON THE DESIRED PORT OR PORTS

FIG. 6

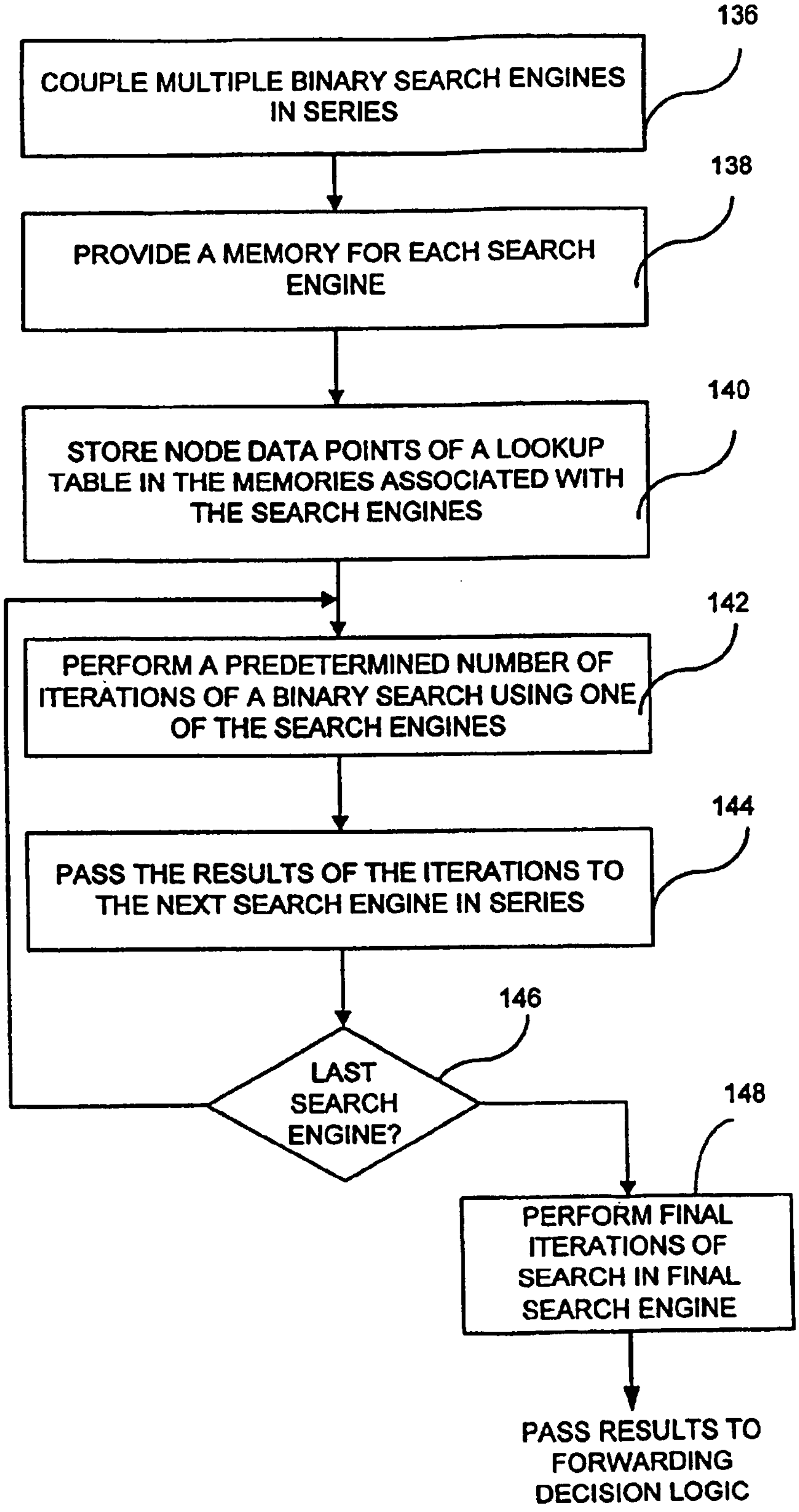

136
COUPLE MULTIPLE BINARY SEARCH ENGINES IN SERIES
138
PROVIDE A MEMORY FOR EACH SEARCH ENGINE
140
STORE NODE DATA POINTS OF A LOOKUP TABLE IN THE MEMORIES ASSOCIATED WITH THE SEARCH ENGINES
142
PERFORM A PREDETERMINED NUMBER OF ITERATIONS OF A BINARY SEARCH USING ONE OF THE SEARCH ENGINES
144
PASS THE RESULTS OF THE ITERATIONS TO THE NEXT SEARCH ENGINE IN SERIES
146
LAST SEARCH ENGINE?
148
PERFORM FINAL ITERATIONS OF SEARCH IN FINAL SEARCH ENGINE
PASS RESULTS TO FORWARDING DECISION LOGIC

FIG. 7

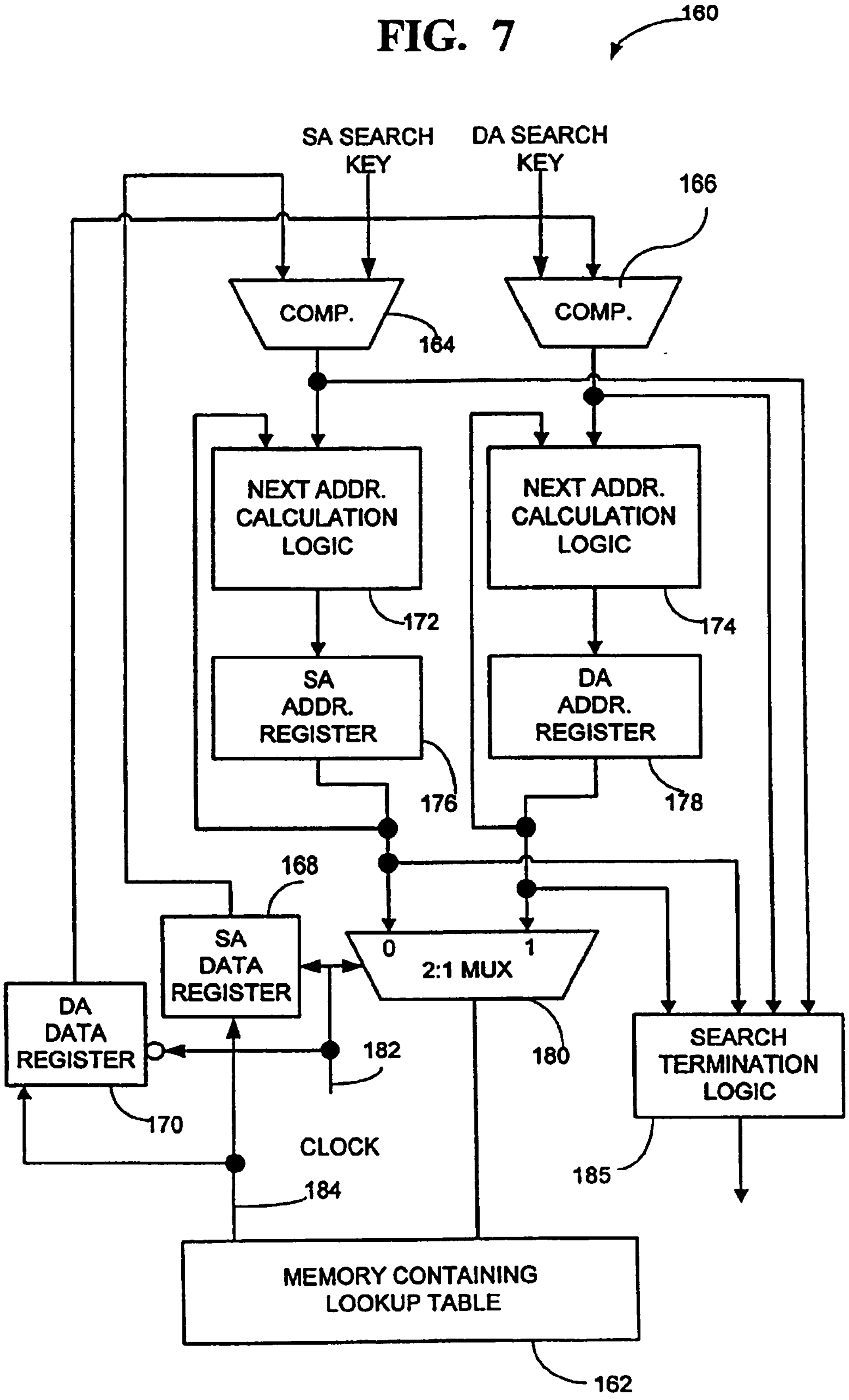

160
SA SEARCH KEY
DA SEARCH KEY
166
COMP.
164
COMP.
NEXT ADDR. CALCULATION LOGIC
172
NEXT ADDR. CALCULATION LOGIC
174
SA ADDR. REGISTER
176
DA ADDR. REGISTER
178
168
SA DATA REGISTER
0
1
2:1 MUX
DA DATA REGISTER
182
180
SEARCH TERMINATION LOGIC
170
CLOCK
185
184
MEMORY CONTAINING LOOKUP TABLE
162

FIG. 8A

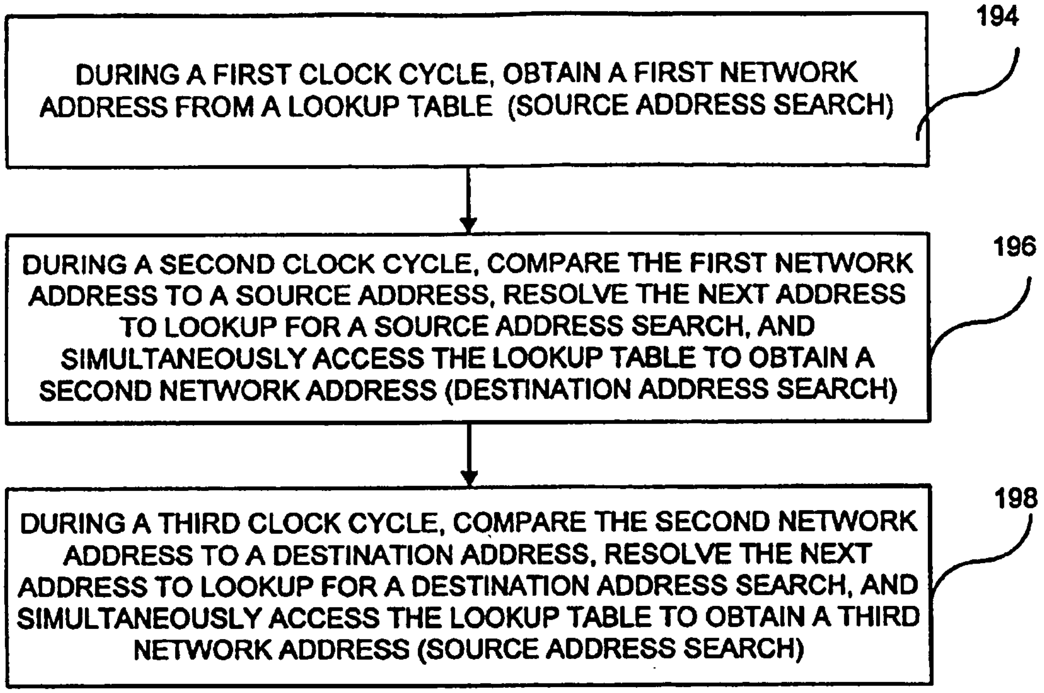

DURING A FIRST CLOCK CYCLE, OBTAIN A FIRST NETWORK ADDRESS FROM A LOOKUP TABLE (SOURCE ADDRESS SEARCH)
194
DURING A SECOND CLOCK CYCLE, COMPARE THE FIRST NETWORK ADDRESS TO A SOURCE ADDRESS, RESOLVE THE NEXT ADDRESS TO LOOKUP FOR A SOURCE ADDRESS SEARCH, AND SIMULTANEOUSLY ACCESS THE LOOKUP TABLE TO OBTAIN A SECOND NETWORK ADDRESS (DESTINATION ADDRESS SEARCH)
196
DURING A THIRD CLOCK CYCLE, COMPARE THE SECOND NETWORK ADDRESS TO A DESTINATION ADDRESS, RESOLVE THE NEXT ADDRESS TO LOOKUP FOR A DESTINATION ADDRESS SEARCH, AND SIMULTANEOUSLY ACCESS THE LOOKUP TABLE TO OBTAIN A THIRD NETWORK ADDRESS (SOURCE ADDRESS SEARCH)
198

FIG. 8B

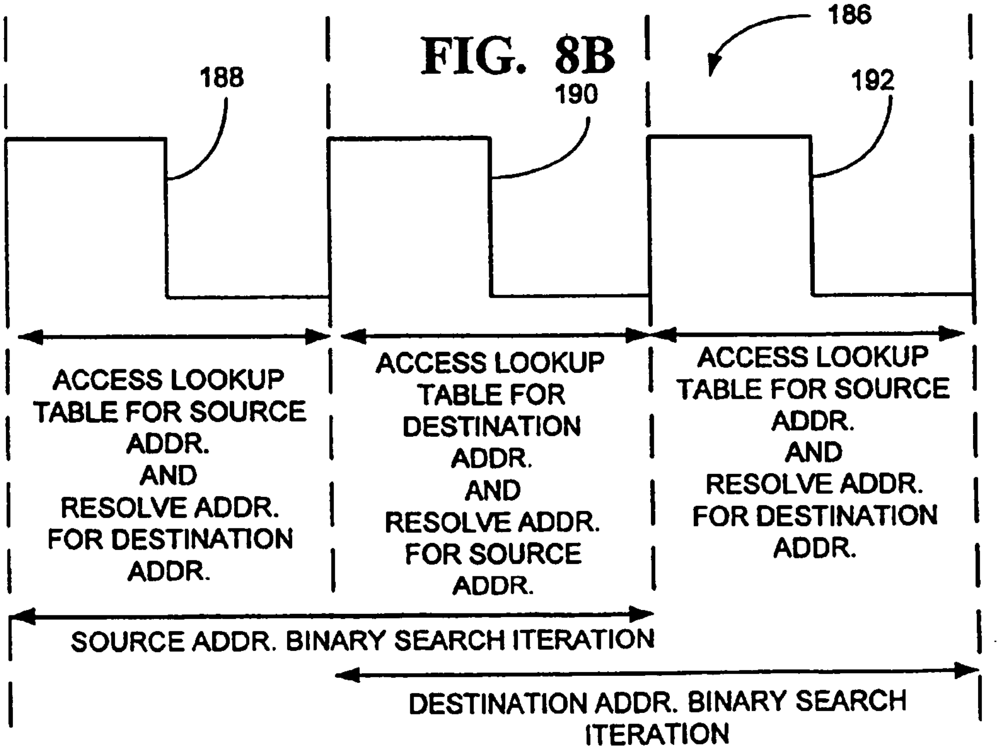
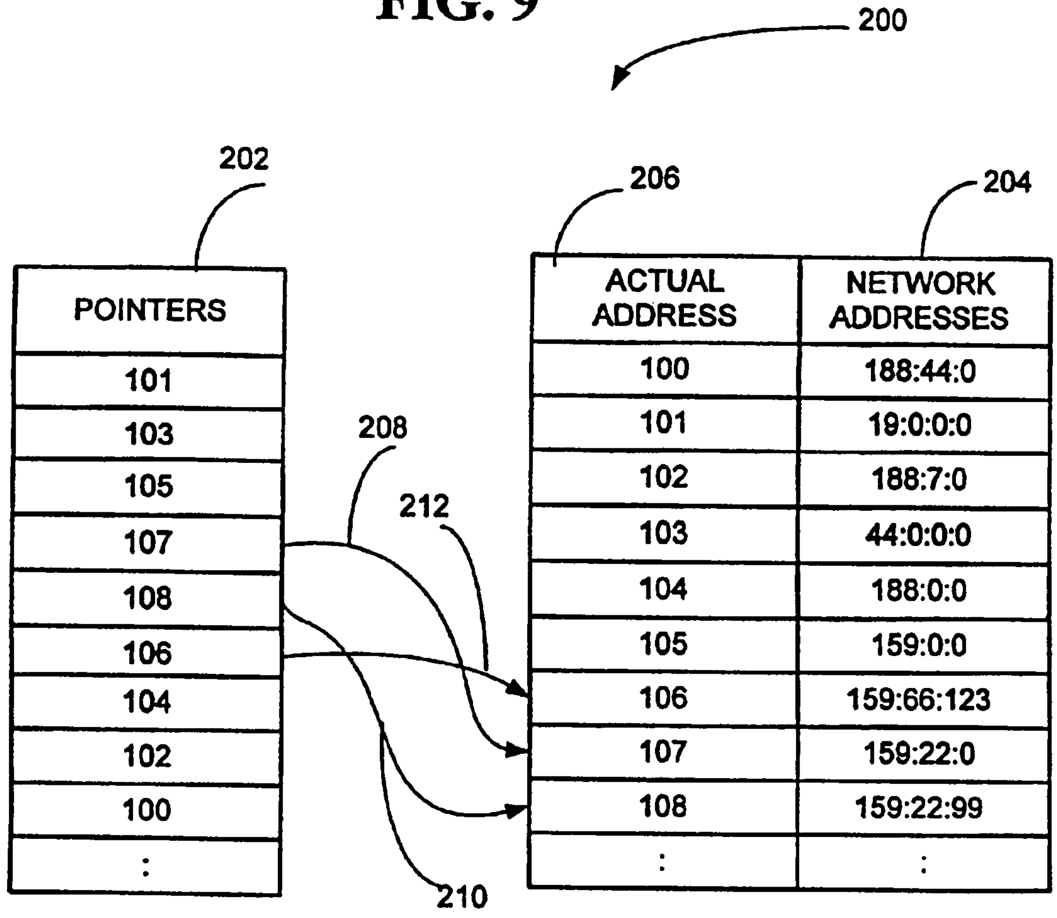

186
188
190
192
ACCESS LOOKUP TABLE FOR SOURCE ADDR. AND RESOLVE ADDR. FOR DESTINATION ADDR.
ACCESS LOOKUP TABLE FOR DESTINATION ADDR. AND RESOLVE ADDR. FOR SOURCE ADDR.
ACCESS LOOKUP TABLE FOR SOURCE ADDR. AND RESOLVE ADDR. FOR DESTINATION ADDR.
SOURCE ADDR. BINARY SEARCH ITERATION
DESTINATION ADDR. BINARY SEARCH ITERATION

FIG. 9



POINTERS
101
103
105
107
108
106
104
102
100
:

ACTUAL ADDRESS	NETWORK ADDRESSES
100	188:44:0
101	19:0:0:0
102	188:7:0
103	44:0:0:0
104	188:0:0
105	159:0:0
106	159:66:123
107	159:22:0
108	159:22:99
:	:

NETWORK SWITCHING DEVICE WITH CONCURRENT KEY LOOKUPS

CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation of prior application Ser. No. 09/166,707, filed Oct. 5, 1998, now U.S. Pat. No. 6,161,144.

This application is based on provisional U.S. patent application Serial No. 60/072,280 filed Jan. 23, 1998, and entitled "Forwarding Database Lookup Technique."

FIELD OF THE INVENTION

This invention relates generally to networks, such as telephone and computer networks, and, more particularly, relates to routing information through such networks.

BACKGROUND OF THE INVENTION

A network allows two or more parties to communicate with each other. In their simplest form, networks generally include transmission lines and switching devices (e.g., routers, switches, switching routers, etc.). The transmission lines carry signals (e.g., electrical, optical, etc.), while the switching devices are intermediate stations that establish temporary connections between transmission lines. In telephone networks, for example, a caller's line goes to a switching device where the actual connection is made to the called party. In computer networks, devices such as routers receive messages on the network and forward the messages to their correct destinations. Computer networks can be as small as a local area network (LAN) consisting of a few computers, printers, and other devices, or it can consist of many computers distributed over a vast geographical area (e.g., the Internet).

An example computer network **10** is shown in FIG. 1A. The network includes two local segments **12** and **14**, and connection to a remote network **16**. Nodes, labeled as A–J, represent computers connected to the local segments. A switching device **20** includes three ports **22–24** and switches network traffic between segments **12**, **14**, and the remote network **16**. Network **16** may also include switching devices, such as switching device 21, which then connects other segments (not shown) to the network. Switching device **20** allows the nodes on one segment to communicate with nodes on other segments and to other switching devices. The nodes communicate with each other through a protocol (e.g., HTTP, TCP/IP, SMB, etc.) which allows the nodes to transmit and receive network frames (a network frame includes a destination address, a source address, and a data field). When switching device **20** receives a frame from a node, it analyzes the destination address by searching a lookup table **26**, shown in FIG. 1B. Lookup table **26** includes table entries having a network address field and a port field. When the destination address is matched to a network address in the lookup table, switching device **20** determines which port to forward the frame to by obtaining the port number corresponding to the matched network address. For example, if node A on segment **12** sends a message to node H on segment **14**, switching device **20** receives the message from node A and in response searches the entries in the network address field of lookup table **26**. Table entry **28** contains the network address for H. A corresponding port field **30** for network address H indicates that the frame should be forwarded over port **2**. Additional background information on switches can be found in a number of references, such as *Fast Ethernet* (1997) by L. Quinn et al., *Computer Networks* (3^{rd} Ed. 1996 by A. Tannenbaum, and *High-Speed Networking* with *LAN Switches* (1997) by G. Held, all of which are incorporated herein by reference.

The switching device can obtain the network addresses for the lookup table in different ways, depending on the particular implementation of the switching device. For example, the switching device may snoop network traffic so that when a frame is received on a port, the switching device determines if the frame's source address is in the table and, if it is not, adds an entry containing the source address and the inbound port to the table. Thus, the switching device is said to "learn" addresses and port numbers from any frame that is transmitted by a node. Another technique some switching devices (e.g., routers) use to obtain the lookup table is from other switching devices through a special protocol. Thus, routers supply network addresses to each other to supplement their lookup tables.

Consequently, when a network frame is received in a switch, both the source and destination addresses must be searched in the lookup table—the source address for "learning" and the destination address for forwarding. To search the lookup table, a single search engine (not shown) within the switch **20** sequentially accesses lookup table entries and compares the entries to the destination address in the network frame. After the search for the destination address is completed, a second independent search is performed for the source address. An example timing diagram **40** for the search engine is shown in FIG. **2**. During a first clock cycle **42**, the search engine accesses the lookup table and obtains a network address. During a second clock cycle **44**, the search engine compares a network address obtained from the lookup table to the destination address. The first and second clock cycles together form a single iteration of the search. If there is no match, the search engine loads an address for the next lookup table entry to analyze. The process is repeated during clock cycles **46** and **48** (a second iteration), and so on until a match is found or the search fails. After the search for the destination address is completed, a second search is performed for the source address. Unfortunately, given current memory and ASIC speed limitations, the above-described technique for searching the lookup table is too slow to meet the requirements of recently-developed gigabit switches. Additionally, with sequential searching, the lookup table is only accessed every other clock cycle, wasting valuable time.

An objective of the present invention, therefore, is to provide a high-speed network switching device that can quickly and efficiently search through address lookup tables and that overcomes the limitations of the prior art.

SUMMARY OF INVENTION

The present invention provides a switching device (e.g., router, switch, switching router, telephone switch, etc.) that forwards network traffic to a desired destination on a network, such as a telephone or computer network. The switching device includes multiple ports and uses a lookup table to determine which port to forward network traffic over. The network traffic is typically in the form of network frames that include source and destination addresses.

In one aspect of the invention, the lookup table includes network addresses that are maintained in sorted order (e.g., ascending or descending order) to facilitate binary searches. Additionally, multiple binary search engines are connected in series and perform multiple binary searches simultaneously. Rather than having each binary search engine perform an independent search, the binary search engines each perform a predetermined number of iterations of an N iteration search. Additionally, each search engine has a separate memory, which stores only data (nodes from the lookup table) necessary for its respective iterations of the search. When a search engine completes its iterations of the search, the results are passed to the next search engine in the series. The next search engine uses the results from the previous search engine as a starting point to performing its respective iterations of the binary search. The number of search engines connected in series (also called pipelining) can vary between two and N, where N is the number of iterations needed for a binary search to complete. If N search engines are used, each search engine performs only one iteration.

By pipelining search engines, increased throughput is achieved. Additionally, by storing only the data necessary for a predetermined number of iterations, very little memory space is needed. For example, a lookup table with 64K entries requires 16 iterations (2^6=64K) to search the entire table using a binary search. If two search engines are used, each search engine performs 8 iterations each. The first search engine's memory only stores 256 entries (2^8), while the second search engine stores either all 64K entries or the remainder of the entries after extraction of the 256 entries from the lookup table.

In another aspect of the invention, each search engine may perform concurrent source and destination searches of the lookup table. That is, during one clock cycle, the search engine performs part of a search for the destination address of the network frame and during the next clock cycle the search engine performs part of a search for a source address. Thus, one search engine performs two independent searches concurrently. Moreover, while the lookup table is being accessed for the destination address, the source address from the previous look up is being analyzed so that a next source address may be calculated. Likewise, when the lookup table is being accessed for the source address, the previously obtained destination address is analyzed. The concurrent source and destination address lookups allow the search engines to search at twice the speed of search engines performing linear searches. Additionally, the lookup table is accessed during every clock cycle rather than sitting idle during some clock cycles. Performing concurrent searches is independent of the algorithm used. Thus, binary, linear, CAM, binary radix trees, hashing and other types of searches can be used.

It will be recognized by those skilled in the art that the two described aspects of the invention can be performed independent of one another. That is, a switching device may have just one binary search engine (as opposed to multiple search engines in series) that performs concurrent source and destination address searches. Additionally, a switching device may have multiple search engines connected in series wherein the search engines do not perform concurrent searches. Alternatively, a switching device may include both aspects with search engines coupled in series and with the search engines performing concurrent searches.

These advantages and other advantages and features of the inventions will become apparent from the following detailed description, which proceeds with reference to the following drawings.

BRIEF DESCRIPTION OF DRAWINGS

FIG. 1A shows a known network containing multiple segments connected through a switching device.

FIG. 1B shows a known lookup table stored in the switching device for identifying nodes connected to the segments.

FIG. 2 shows a timing diagram of a search engine within the switching device of FIG. 1A.

FIG. 3 shows a switching device according to one aspect of the invention with two binary search engines coupled in series.

FIG. 4A shows a lookup table in sorted order according to another aspect of the invention.

FIG. 4B shows the lookup table of FIG. 4A in binary tree format and showing four iterations of a binary search.

FIG. 4C shows a memory configuration for a switching device having four search engines with each search engine performing one of the iterations of FIG. 4B.

FIG. 4D shows search results that are passed between search engines from binary searches performed on the lookup table of FIG. 4A.

FIG. 5 shows a flowchart of a method for forwarding network frames in the switching device of FIG. 3.

FIG. 6 shows a flowchart of a method for performing different iterations of a binary search in separate binary search engines.

FIG. 7 shows a detailed circuit diagram of a binary search engine according to another aspect of the invention where source and destination searches are performed concurrently.

FIG. 8A shows a flowchart of a method for concurrently performing source and destination searches using the binary search engine of FIG. 7.

FIG. 8B is a timing diagram for the binary search engine of FIG. 7.

FIG. 9 shows another example of a lookup table using pointers to access the network addresses.

DETAILED DESCRIPTION OF AN ILLUSTRATIVE EMBODIMENT

FIG. 3 shows a switching device **50** having a port **52** and port intercommunication logic **54**, Port **52** includes a media interface **56**, a primary memory **58**, and a search engine **60**. The search engine **60** includes temporary packet storage **62**, packet analysis and key extraction logic **64**, two internal binary search engines **66**, **68**, a first stage memory **70**, and forwarding decision logic **72**. There are multiple ports (not shown) in switching device **50**. One or more ports are located on channel cards (not shown) mounted in a chassis. The number of ports and how the ports and port intercommunication logic **54** are mounted within a chassis are based on the particular application and are not important to the invention.

Media interface **56** connects switching device **50** to a network (not shown) through a network cable **74**. The network cable can take a variety of forms (e.g., fiber optic, twisted-pair, coaxial, etc.) depending on the type of network. A variety of network standards and protocols may be used, such as TCP/IP, IPX/SPX, FDDI, ATM, ETHERNET, GIGABIT ETHERNET, FAST ETHERNET, Token Ring, SONET, 100-base TX, etc. Other network protocols, standards, and network cables now existing or later developed may be used with the invention, since these particular aspects are not important to the invention. Media interface **56** is a communication link between search engine **60** and the network. Thus, media interface **56** allows search engine **60** to send network frames in any desired format and media interface **56** reformats the frames for the particular network.

Similarly, media interface **56** receives network frames from the network and formats the frames so they may be read by search engine **60**. The media interface used is based on the particular application and is not important to understanding the invention.

Temporary packet storage **62** within search engine **60** holds the network frame temporarily while other components within the search engine determine where to forward the network frame to, as is further described below. In some circumstances, a decision may be made to not forward the packet at all. In such cases, the network frame is never passed to the port intercommunication logic. Depending on the size of the network frame, the temporary packet storage **62** may hold several network frames, or, alternatively, only a portion of a network frame as it is forwarded to another port.

Packet analysis and key extraction logic **64** extracts the source and destination addresses from the network frame and forwards the addresses to the first binary search engine **66**. The packet analysis and key extraction logic **64** may also pass additional information to the binary search engine **66**, such as virtual LAN information that comes with the network frame or is derived based on the type of network frame. This additional information is appended to the source and destination addresses to form a key. One skilled in the art will recognize that the key may include any desired information, depending on the particular application. For example, the key may contain only the destination address, or only the source and destination addresses, or the source and destination addresses and additional information. For simplicity, the key is described generically below as being the source and destination addresses.

Binary search engines **66** and **68** are coupled to respective memories **58** and **70**. The memories together store a lookup table that the binary search engines use for analyzing network frames received from media interface **56**. A portion of an example lookup table is shown in FIG. **4** and is further described below. A management processor (not shown) is coupled to search engines **66**, **68** and maintains the lookup table in memories **58**, **70**. Specifically, the management processor may direct the search engines to delete old table entries, insert new table entries and generally maintain the table in sorted order so that the search engine **60** performs searches efficiently and correctly.

As is described in detail below, binary search engines **66**, **68** perform a binary search on the lookup table. A binary search is a technique for searching a sorted table where successive midpoints of the table are determined and compared against a search term. Thus, to start the search the midpoint of the table is compared against the search term. By using the midpoint, the table is effectively divided into two parts and a determination is made as to which of the two parts the search term must reside in. Only the part that has the search term is analyzed and the other part is ignored. The midpoint of the pertinent part is then determined to divide the table into two more parts, and so on until the key is found or the search fails. Binary searches are understood in the art and are described in a text entitled *Design and Analysis of Algorithms* by Jeffrey D. Smith, 1989. Binary searches require a predetermined number of "iterations" to complete. The number of iterations depends on the size of the lookup table and follows the formula $\log_2 N$=number of iterations, where N is the number of entries in the table. Thus, a 64K table requires 16 iterations to complete a binary search, while a 256 entry table requires 8 iterations to complete.

One aspect of the invention is that the binary search engines divide the binary search of the lookup table by each performing some of the iterations of the overall search. For example, if the lookup table has 64K entries, the binary search engine **66** performs the first eight iterations of the search and binary search engine **68** performs the last eight iterations. Additionally, the first stage memory **70** does not contain the entire lookup table. Instead, it only contains 256 entries needed for the first eight iterations of the search. The 256 entries are determined and copied from the lookup table and include the successive midpoint combinations from the lookup table needed for the first eight iterations. Once the binary search engine **66** completes its eight iterations, the results are passed to binary search engine **68**. Binary search engine **68** then uses the results as a starting point for its eight iterations of the binary search. A substantial amount of memory is saved by only using successive midpoint combinations in the memory **70**. Additionally, although two search engines are shown, any desired number of search engines may be used. For example, for a 16 iteration search, 16 binary search engines can be used with each search engine performing one iteration. Alternatively, 4 binary search engines may be used with each binary search engine performing 4 iterations. Still further, the partitioning of iterations across search engines need not be equal. Also, the lookup table can be any desired length. Consequently, the binary search engines can perform any number of iterations depending on the particular application.

After binary search engine **68** completes the binary search, the results are passed to forwarding decision logic **72**, which examines the results and applies a predetermined set of rules to determine whether the network frame should be forwarded and which port or ports it should be forwarded to. Forwarding decision logic may also examine the level of priority of the network frame. Higher priority frames are typically forwarded by the switching device **50** before lower priority frames.

When the forwarding decision logic **72** determines that a frame is to be forwarded to other ports in switching device **50**, it passes the network frame to the port intercommunication logic **54**. Port intercommunication logic **54** includes a switch fabric **76** and a switch fabric control **78**. Switch fabric **76** can take a variety of forms. For example, the switch fabric can be a cross-bar switch, which is commonly used in telecommunications switching. The cross-bar switch creates a path between a receiving port and a transmitting port so that the network frame may be passed therebetween. A wide variety of cross-bar switches may be used, such as cut-through switches, interim cut-through switches, and store-and-forward switches. Other types of switch fabrics may also be used. For example, switch fabric **76** may also be a central memory using a bus arbitration device and a central bus. Using a shared-memory bus architecture, all ports access a central located memory pool. The ports can access the central memory through a common bus when an arbitration device grants access. Another possible switch fabric that can be used is a parallel access shared-memory architecture. In a parallel access shared memory, all ports share a central memory location. However, a bus arbitration scheme is not used. Instead, every port has a dedicated path into and out of the central memory fabric. Therefore, all ports can simultaneously access the centralized memory pool at any time. A wide variety of existing switch fabrics or later developed switch fabrics may also be used. The particular switch fabric and switch fabric control is not of importance to the invention.

Switch fabric control **78** controls network frames as they are passed through the switch fabric. In the case where the switch fabric is a cross-bar switch, the switch fabric control is typically called a scheduler. The scheduler establishes a connection within the cross-bar switch so that a search engine on one port can directly pass a network frame to a search engine on another port. In the case where the switch fabric is a memory, the switch fabric control tells a receive-side search engine where to store the frame in memory. After the frame is stored in memory, the switch fabric control signals a transmitting-side port that the network frame is ready to be transmitted and provides the address of the memory location where the frame in located. The switch fabric control may also provide priority information to the transmitting-side port.

FIG. 5 shows a flow chart of a method used by the switching device **50** for forwarding network frames. In step **80**, the media interface **56** receives a network frame from the network and passes the frame to search engine **60**. The network frame contains a destination address that indicates the ultimate destination for the network frame. In step **82**, search engine **60** searches memories **58, 70** to determine whether the destination address from the network frame is located within the lookup table. If a network address matches the destination address, a port that the frame should be forward to is also obtained from the lookup table. Assuming that the destination address was properly found in the lookup table and the appropriate port was determined, the search engine passes the search results to switch fabric control **78** (step **84**). In step **86**, search engine **60** transfers the network frame through switch fabric **76**. If the switch fabric is a cross-bar switch, the switch fabric control **78** establishes the connection in the switch fabric and communicates to search engine **60** when to send the network frame. If the switch fabric is a central memory with or without bus arbitration, the switch fabric control tells search engine **60** where in the switch fabric to store the network frame. Search engine **60** then stores the network frame at the indicated location. The switch fabric control also informs the other ports of where the network frame is stored so that they may properly access it within the switch fabric. Regardless of the technique used with the switch fabric, the network frame is obtained from the switch fabric and transmitted on one or more ports (step **88**).

FIG. 4A shows an example lookup table **100** having 16 data entries, 0000B through 1111B. Each table entry includes two or more fields including, an address field **102**, illustrated as a network address, and a forwarding information field **104**, illustrated as a port number. Although a network address field is shown, other addresses used in switching devices may be used. The forwarding information field also may contain information other than the port number. For example, the forwarding information field may analyze information related to the protocol format (e.g., MAC) that the address is in to determine which port to forward the network frame to. The network address field **102** contains network addresses in sorted order (e.g., ascending or descending order). The port number field **104** contains the port associated with the network address field. One or more of these data fields may be deleted, or alternatively, additional fields may be used. Additional fields may, for example, show a priority status associated with the network address. The network address field **102** identifies a destination to transmit the network frame to, such as a computer on a network. The network address field is used as a key for accessing the other fields (often called a payload) in a table entry. Address 0000B is empty so that the lookup table contains an odd number of entries. The midpoint of the table is at address 1000B and the lookup table can be equally divided into two portions **106** and **108** with equal numbers

of data entries. The first portion **106** can similarly be divided into two portions **110, 112** with a midpoint at address 0100B. The second portion **108** also can be divided into two portions **114, 116** with a midpoint at address 1100B. By determining successive midpoints, a binary search tree can be generated.

FIG. 4B shows a binary search tree **118** for the lookup table **100** of FIG. 4A. The search tree shows four iterations, which are the maximum number of iterations required for any search on a 16 entry table (i.e., $2^4=16$). The tree is formed by determining the successive midpoints, called nodes, in the lookup table as described in relation to FIG. 4A. To perform a binary search, the search term is compared against the network address located at the midpoint address 1000B of the lookup table (i.e., 26). If the search term is less than 26, it is compared to 14. Alternatively, if it is greater than 26 it is compared to 38, and so on, until a match is found or the search fails. Each comparison that is performed is another iteration of the binary search.

FIG. 4C shows that the different iterations of the binary search may be divided amongst 4 different binary search engines connected in series. A memory **120** for a first search engine only requires one data entry (one node) for the first iteration of the binary search, which represents the midpoint of the lookup table. A memory **122** for a second search engine requires two data entries for the second iteration, which represent the midpoints of portions **106** and **108** of the lookup table **100** (See FIG. 4A). A memory **124** for a third search engine contains additional successive midpoints of the lookup table needed for the third iteration. And a memory **126** for a fourth search engine contains the remaining data entries of the lookup table. The fourth memory **126** may also contain the entire lookup table **100**. The memories **120, 122,** and **124** (called the precursor memories) may include only the network address field of the lookup table, without any payload fields. Alternatively, the precursor memories **120, 122,** and **124** may include the payload information so that the lookup table is effectively distributed amongst many memories, even though it is a unitary database. In any event, the precursor memories contain successive midpoint possibilities (nodes) for the different iterations of a binary search and the nodes are stored adjacent to one another. The precursor memories could contain additional table entries other than just the successive midpoint possibilities, if desired.

FIG. 4D shows an example of data passed between the four search engines of FIG. 4C in searching for the network address 17 (the key) located at address 0101 in the lookup table (see FIG. 4A). As comparisons are made during iterations of the binary search, the final address of 0101 is generated. That is, the first search engine generates the most significant bit, the last search engine generates the least significant bit, and the intermediate two bits are generated by the intermediate search engines. When a comparison is made, if the key is less than the network address obtained from the lookup table, a zero is passed to the next search engine. Alternatively, if the key is greater than the network address, a one is passed to the next search engine. Thus, with the key being 17, the first search engine compares 26 (the first iteration) and 17. Since 17 is less than 26, a 0 is passed to the next search engine in series as indicated at **128**. The 0 represents the most significant bit of the final address where the network address 17 resides in the lookup table. The next search engine then compares 17 against 14, since 38 is no longer a possibility. Since 17 is greater than 14, the next bit in the address is a 1. Consequently, a 1 is appended to the most significant bit and a 01 is passed to the next search engine as indicated at **130**. The next search engine then compares 17 and 20. Since 17 is less than 20, a 0 is appended to the previous passed address resulting in a 010 being passed to search engine **4**, as indicated at **132**. The last search engine must then compare 17 against 17 to obtain an address of 0101, which is the appropriate address in the lookup table (See FIG. **4A** at **134**).

FIG. **6** shows a flowchart of a method for performing binary searches as described in FIGS. **4A–4D**. In step **136**, multiple search engines are coupled in series. Virtually any number of search engines may be used. For example, FIG. **3** shows two search engines, while FIG. **4C** describes using four search engines. The invention can easily be extended to eight, sixteen, or any other number of search engines. Each search engine that performs iterations of the binary search has its own separate memory (step **138**). The memories for the search engines store nodes (the network address field with or without the payload information) needed for that search engine to perform its respective iterations of the binary search (step **140**). The nodes are the successive midpoint possibilities of the lookup table in binary tree format. In step **142**, a first search engine in the series performs a predetermined number of iterations of the binary search. The results of the iterations are then passed to the next search engine in the series (step **144**). In step **146**, a check is performed to see whether the results are being passed to the last search engine in the series. If not, then steps **142** and **144** are repeated. If yes, then the final iterations of the binary search are performed and the results are passed to the forwarding decision logic (step **148**).

Although a wide variety of binary search engines can be used in the method and apparatus described in FIGS. **1–6**, a particular search engine is shown in relation to FIGS. **7**, **8A**, and **8B**. This search engine can also be used independently of the method and apparatus of FIGS. **1–6**. In particular, the search engine can be used with searching techniques other than binary searches including linear searches, CAM, binary radix trees, hashing, etc.

Prior search engines perform a search for the source address from the network frame and, only when the source address search is completed, perform a search for the destination address (See FIG. **2**). The present invention, by contrast, performs both source and destination address searches concurrently. FIG. **7** shows a search engine **160** coupled to a memory **162** containing a lookup table. The search engine **160** includes a source address comparator **164** and a destination address comparator **166**. The source address comparator **164** has one input coupled to a source address (SA) search key. This SA search key is the source address obtained from the network frame and passed to the binary search engine **160** by the packet analysis and key extraction logic **64** (see FIG. **3**). A second input to comparator **164** is coupled to a source address data register **168** that contains a network address most recently obtained from memory **162** during a previous iteration of the source address search. The destination address comparator **166** also has two inputs with one coupled to a destination address (DA) search key obtained from the network frame and the other input coupled to a destination address data register **170** containing a network address obtained from the memory **162** during a previous iteration of the destination address search. The output of the comparators **164**, **166** are coupled to respective next address calculation logic circuits **172**, **174**. The next address calculation logic circuits **172**, **174** use the lookup table address from the previous search iteration and the results of the comparators **164**, **166** to determine the next lookup table address to analyze. The logic circuits **172**, **174** make such a determination of the next address based on the nodes (i.e., successive midpoints) of the lookup table in the case of a binary search. In the case, of a linear search, the next address is simply the previous address plus one. Thus, the logic circuits **172**, **174** perform different functions based on the application. The outputs of logic circuits **172**, **174** are coupled to respective SA and DA address registers **176**, **178**. The outputs of the address registers **176**, **178** are coupled to multiplexer **180** and are also fed back into next address calculation logic **172**, **174**, respectively. The multiplexer **180** is coupled to a clock line **182** that switches a multiplexer output so that the contents of the SA address register **176** and the DA address register **178** are passed to memory **162** in an alternating fashion. The contents of the SA and DA address registers **176**, **178** contain an address of a lookup table entry and when applied to memory **162** causes the memory to output data (i.e., a network address) on a data output **184**. Clock **182** is also coupled to both data registers **168**, **170** and is synchronized so that data is latched into source data register **168** when the multiplexer **180** passes the contents of source address register **176**. Similarly, data register **170** latches the data output **184** of memory when the address stored in DA address register **178** passes through the multiplexer **180**. Search termination logic **185** determines when a search is complete by analyzing outputs of the address registers **176**, **178** and the comparators **164**, **166**. An output of the search termination logic **185** can be passed to the forwarding decision logic **72** (see FIG. **2**) or to another search engine connected in series to signal when a search is complete and to pass the results of the search.

FIG. **8B** shows a timing diagram for the circuit of FIG. **7**. A clock signal **186** is at twice the frequency of the clock on clock line **182**. Three cycles **188**, **190**, and **192** of clock signal **186** are shown. During a first clock cycle **188**, the SA address register **176** (FIG. **7**) is already loaded with a lookup table address and the multiplexer **180** passes that address to the memory **162**. Consequently, memory **162** outputs data (a network address) on data output **184** corresponding to the address in the SA address register. At the end of cycle **188**, the source data register **168** is loaded with the data from the memory **162**. Also during cycle **188**, logic circuit **174** is calculating the next lookup table address to analyze for the destination address search and the results are stored in DA address register **178**. Clock cycle **188** represents the start of a source address search iteration.

During clock cycle **190**, comparator **164** compares the SA search key and the SA data register **168**. The next address calculation logic **172** determines the next lookup table address to analyze for the search, and the results are loaded into the SA address register **176**. Also during clock cycle **190**, the output of the DA address register is passed through multiplexer **180** and the corresponding data from memory **162** is latched into DA data register **170**. Clock **190** represents the start of a destination address search iteration. At the end of clock cycle **190**, the source address search iteration is completed. Thus, clock cycles **188**, **190** represent a complete source address search iteration in a search requiring multiple iterations.

During clock cycle **192**, another source address search iteration is started and the destination address search iteration is completed. The comparator **166** compares the DA search key and the output of the DA data register **170**. The next address calculation logic **174** calculates the next lookup table address to examine and loads the calculated address into the DA address register. Thus, during the three clock cycles **188**, **190**, and **192** both a source and destination address search iterations are performed concurrently. Additionally, the source and destination address searches are independent of each other.

The hardware of FIG. 7 can be modified while maintaining the same timing diagram of FIG. 8B. For example, the multiplexer **180** can be removed and the address registers **176, 178** can have tristate outputs with the clock line **182** coupled to the address registers **176, 178**. Thus, the clock switches the address registers **176, 178** in alternating fashion onto a common bus attached to memory **162**. Other modifications can also be made to the hardware, but the general timing diagram of FIG. **8B** should remain the same.

FIG. **8A** is a flowchart summarizing the steps for concurrently performing source and destination address searches. During a first clock cycle, a network address is obtained from the lookup table as a first phase of a source address iteration (step **194**). During a second clock cycle (step **196**), the network address is compared to the SA search key. The next address calculation logic **172** reads the comparator **164** and determines what lookup table address should be accessed next. Also during the second cycle, the lookup table is accessed for the next network address needed for a first phase of a destination address iteration and the source address iteration is completed. Finally, during a third cycle (step **198**), the destination address iteration is completed by resolving the next network address to analyze for the destination address search. Also, the first phase of the source address iteration is repeated. Although only three clock cycles are shown, the clock is continuous.

A wide variety of searching techniques can be used with the concurrent search embodiment including binary, linear, CAM, binary radix trees, hashing and other types of searches.

FIG. **9** shows another example of a lookup table **200** that may be used according to the invention. The lookup table includes a list of pointers **202** and network address data **204** stored in the memory within the switching device. The pointers point to the network address data **204**. An example of actual memory addresses is shown in column **206** for illustrative purposes. Notably, neither the network addresses **204** nor the list of pointers **202** are stored in sorted order. However, the contents of data pointed to by the list of pointers **202** are stored in sorted order. For example, a pointer **208** points to address **107**, which contains network address 159:22:0. A next pointer **210** in the list points to address **108** which contains the network address 159:22:99. The next pointer **212** points to address **106** which contains the network address 159:66:123. Although the list of pointers **208, 210**, and **212** are not in order (since the pointers are **107, 108, 106**, respectively) the data pointed to by the pointers is in order since addresses 159:22:0, 159:22:99 and 159:66:123 are in ascending order.

Thus, the lookup table according to the invention does not need to have data stored in contiguous addresses. Instead, network addresses may be in one memory location and forwarding information (such as port numbers) may be stored in other locations of memory. Pointers or some other technique for linking associated data is then used.

Having described and illustrated the principles of our invention with reference to preferred embodiments thereof, it will be apparent that these embodiments can be modified in arrangement and detail without departing from the principles of the invention.

For example, although the port is shown as including separate components, such components can be formed in a single integrated circuit. Additionally, other circuit components, such as the port intercommunication logic **54** can also be included in the same integrated circuit as the port **52**.

Additionally, any of the components of the switching device can be performed by hardware, software, or a combination thereof. The invention should not be limited to the particular technique (whether hardware or software) for carrying out the methods and apparatus described herein. For example, the search engine can be a microprocessor running software or an ASIC where the searches are performed in hardware.

Further, although the search engines are generally shown performing equal iterations of binary searches, it is not necessarily so. That is, the search engines can perform unequal iterations of searches. For example, in a 16 iteration search, a first search engine can perform 9 iterations and a second search engine can perform 7 iterations. Other variations in the number of iterations can easily be applied and depends on the particular application and number of search engines used.

Still further, although the search engine of FIG. 7 shows two comparators and two next address calculation logic circuits, one of each may be used instead, with the clock controlling whether the SA address register or the DA address register is read by the next address calculation logic circuit. Additionally, the clock controls the single comparator so that only one of the DA data register and SA data register are compared at a time to one of the SA and DA search keys.

Still yet further, although the lookup table is shown as having all memory locations filled, in practice this is not generally the case. Instead, the memory has only a portion filled with lookup table values and the remaining portion is padded with dummy values. For example, the table of FIG. **4A** may have only 9 lookup table entries with the remaining memory locations containing dummy values. In the case of an ascending table, the dummy values are a number sufficiently large enough that no network address can be confused with them. In the case of a descending table, the dummy values are a number sufficiently small enough that no network address can be confused with them. Additionally, the lookup table of FIG. **4A** contains only 16 entries for illustration. In practice, lookup tables contain hundreds of thousands of entries.

Still further, although the lookup table is shown as containing table entries with multiple fields, the lookup table may be a list of pointers, and the pointers point to the network addresses and other fields. Additionally, the lookup table can be any type of data structure or array that store data.

Yet further, the network packets can include any kind of data including video images, voice data during a phone call, a document, etc.

Still further, when a port receives a network frame, it can analyze the destination address (at layer two and layer three) and it can also analyze layer **4** policy information if desired.

Still yet further, the present invention may be applied to LAN'S, WAN's, the Internet, Intranets, telephone networks, or any other network.

Additionally, although the lookup table is described as containing network addresses, the lookup table may include other keys. A key may include a network address exclusively or in combination with additional information. Alternatively, a key may include lookup information other than network addresses. Additionally, the switching device may receive network frames that include search keys. Alternatively, the search key may be derived from the network frame. For example, the search engine may use characteristics of the network frame, such as what port it was received on or what protocol format it is in, to derive the search key. The search key, which is either explicitly included in the network frame or derived therefrom, is compared to the lookup table which includes lookup keys.

In view of the many possible embodiments to which the principles or invention may be applied, it should be recognized that the illustrated embodiment is only a preferred example of the invention and should not be taken as a limitation on the scope of the invention. Rather, the invention is defined by the following claims. We therefore claim as the invention all such embodiments that come within the scope of these claims.

We claim:

1. A network switching device comprising:

a first search engine;

a first memory coupled to the first search engine;

a second search engine;

a second memory coupled to the second search engine characterized in that the first search engine performs a search of the first memory for a first key concurrently with a search of a second key and passes results of the search for the first and second keys to the second search engine for concurrently searching the second memory for the first and second keys.

2. The network switching device of claim 1, wherein the first key is a source address and the second key is a destination address.

3. The network switching device of claim 1, wherein each search engine accesses its respective memory during successive clock cycles and alternates between accessing the memory for search of the first key and search of the second key.

4. The network switching device of claim 1, wherein each search engine performs iterations of searches of the first and second keys in three clock cycles, wherein during at least one of the three clock cycles an iteration of the search for the first key is completing while an iteration of the search for the second key is beginning such that there is an overlapping between execution of the iterations of the searches of the first and second keys.

5. The network switching device of claim 1, wherein the first and second search engines together perform a search of N iterations, and wherein the first search engine performs a first portion of the N iterations and the second search engine performs second portion of the N iterations.

6. The network switching device of claim 5, wherein the first memory only includes a first portion of a lookup table necessary to perform the first portion of the N iterations and the second memory only includes a second portion of the lookup table necessary to perform the second portion of the N iterations.

7. The network switching device of claim 1, wherein the first and second search engines are binary search engines.

8. In a network switching device, a method for searching a lookup table comprising the steps of:

receiving a data packet including a first key and a second key;

extracting the first and second keys from the data packet;

performing by a first search engine a search of a first memory for the first key;

performing by the first search engine a search of the first memory for the second key concurrently with the search for the first key; and

passing results of the search for the first and second keys to a second search engine coupled to the first search engine for concurrently searching the second memory for the first and second keys.

9. The method of claim 8, wherein the first key is a source address and the second key is a destination address.

10. The method of claim 8, wherein each search engine accesses its respective memory during successive clock cycles and alternates between accessing the memory for search of the first key and search of the second key.

11. The method of claim 8, wherein each search engine performs iterations of searches of the first and second keys in three clock cycles, wherein during at least one of the three clock cycles an iteration of the search for the first key is completing while an iteration of the search for the second key is beginning such that there is an overlapping between execution of the iterations of the searches of the first and second keys.

12. The method of claim 8, wherein the first and second search engines together perform a search of N iterations, and wherein the first search engine performs a first portion of the N iterations and the second search engine performs a second portion of the N iterations.

13. The method of claim 12, wherein the first memory only includes a first portion of a lookup table necessary to perform the first portion of the N iterations and the second memory only includes a second portion of the lookup table necessary to perform the second portion of the N iterations.

14. The network switching device of claim 1, wherein the first and second search engines are binary search engines.

* * * * *

US006389462B1

(12) **United States Patent**
Cohen et al.

(10) **Patent No.: US 6,389,462 B1**
(45) **Date of Patent: May 14, 2002**

(54) **METHOD AND APPARATUS FOR TRANSPARENTLY DIRECTING REQUESTS FOR WEB OBJECTS TO PROXY CACHES**

(75) Inventors: **Ariel Cohen**, Berkeley Heights; **Sampath Rangarajan**, Bridgewater; **Navjot Singh**, Morristown, all of NJ (US)

(73) Assignee: **Lucent Technologies Inc.**, Murray Hill, NJ (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/212,980**

(22) Filed: **Dec. 16, 1998**

(51) **Int. Cl.**^{7} **G06F 15/16**

(52) **U.S. Cl.** **709/218**; 709/201; 709/203; 709/219; 709/229; 709/231; 709/239; 707/10; 707/501

(58) **Field of Search** 709/218, 203, 709/219, 201, 224, 229, 239, 231; 707/10, 501; 713/201

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,774,660 A	*	6/1998	Brendel et al.	709/201
5,838,916 A	*	11/1998	Domenikos et al.	709/219
5,864,852 A	*	1/1999	Luotonen	707/10
5,905,872 A	*	5/1999	DeSimone et al.	709/245
5,941,954 A	*	8/1999	Kalajan	709/239

(List continued on next page.)

OTHER PUBLICATIONS

ACEdirector 2 Data Sheet, ACElerate Layer 4 Services, Server Switching Services, http;//www.alteon.com.

"Deploying Transparent Caching With Inktomi's Traffic Server", http://www.inktomi.com.

Danzig, Peter, Network Appliance, Inc., "NetCache Architecture and Deployment", http://www.network application.com.

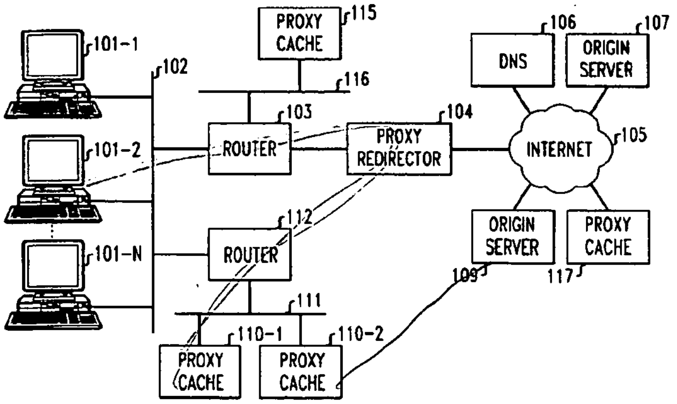
(List continued on next page.)

Primary Examiner—Ayaz Sheikh
Assistant Examiner—Frantz B. Jean
(74) *Attorney, Agent, or Firm*—Stephen M. Gurey

(57) **ABSTRACT**

In order to transparently redirect an HTTP connection request that is directed to an origin server (107) to a proxy cache (110-1), a proxy redirector (104) translates the destination address of packets directed to the origin server to the address of the proxy. During a handshaking procedure, a TCP connection is transparently established between the client (110-1) and the proxy cache. When the client transmits a GET request to what it thinks is the origin server, which request specifies the complete address of an object at that origin server that it wants a copy of, the proxy redirector modifies the complete address specified in that GET request before it is sent to the proxy cache. Specifically, the IP address of the origin server found in the destination field in the IP header of the one or more packets from the client containing the GET request is added by the proxy redirector as a prefix to the complete URL in the GET request to form an absolute URL. The proxy cache determines from that absolute URL whether it has the requested object stored in its cache. If it does, it sends the object back to the proxy redirector, which masquerades those packets as coming from the origin server by translating their destination address to the address of the client and their source address to that of the origin server. If the proxy does not have the requested object, a separate TCP connection is established between the proxy and the origin server from where the object is retrieved and then forwarded over the TCP connection between the client and the proxy. In order to account for the additional number of bytes in the GET request, an acknowledgement sequence number in packets returned from the proxy that logically follow receipt of the GET request are decremented by that number by the proxy redirector before being forwarded to the client. Similarly, a sequence number in packets transmitted by the client subsequent to the GET request are incremented by that number before being forwarded by the proxy redirector to the proxy cache.

55 Claims, 4 Drawing Sheets



U.S. PATENT DOCUMENTS

5,987,523	A	*	11/1999	Hind et al.	709/245
6,065,058	A	*	5/2000	Hailpern et al.	709/231
6,081,829	A	*	6/2000	Sidana	709/203
6,081,900	A	*	6/2000	Subramaniam et al.	713/201
6,098,108	A	*	8/2000	Sridhar et al.	709/239
6,112,212	A	*	8/2000	Heitler	707/501
6,138,162	A	*	10/2000	Pistriotto et al.	709/229
6,189,030	B1	*	2/2001	Kirsch et al.	709/224

OTHER PUBLICATIONS

"High–Performance Web Caching White Paper", CacheFlow, Document Center, http://www.cacheflow.com.

"Shared Network Caching and Cisco'Cache Engine", http://www.cisco.com.

"The Content Smart Internet", ArrowPoint, ArrowPoint Communications, 235 Littleton Road, Westford, MA 01886, http://www.arrowpoint.com.

Danzig, P. and Swartz, K. L., "Transparent, Scalable, Fail–Safe Web Caching", Network Appliance, Inc., http://www.networkappliance.com.

RND Networks Inc., "Cache Directing Technology—White Paper", http://www.rndnetworks.com.

Foundry Products, "ServerIron Server Load Balancing and Transparent Caching Swithch", http://www.foundrynet.com.

* cited by examiner

FIG. 1

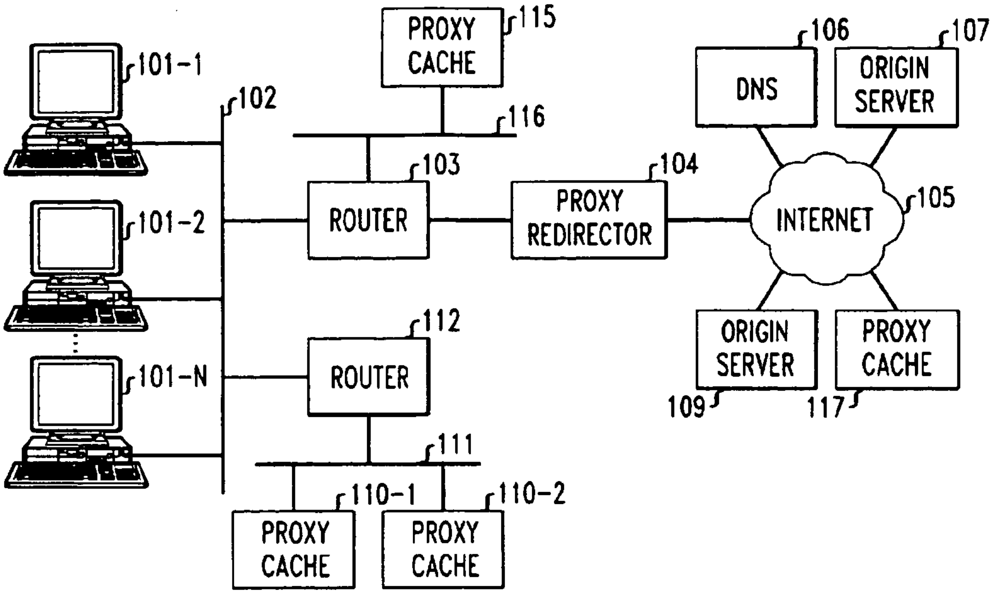

PROXY CACHE
115
101-1
102
116
106
DNS
107
ORIGIN SERVER
103
ROUTER
104
PROXY REDIRECTOR
INTERNET
105
101-2
112
ROUTER
ORIGIN SERVER
PROXY CACHE
109
117
101-N
111
110-1
110-2
PROXY CACHE
PROXY CACHE

FIG. 3

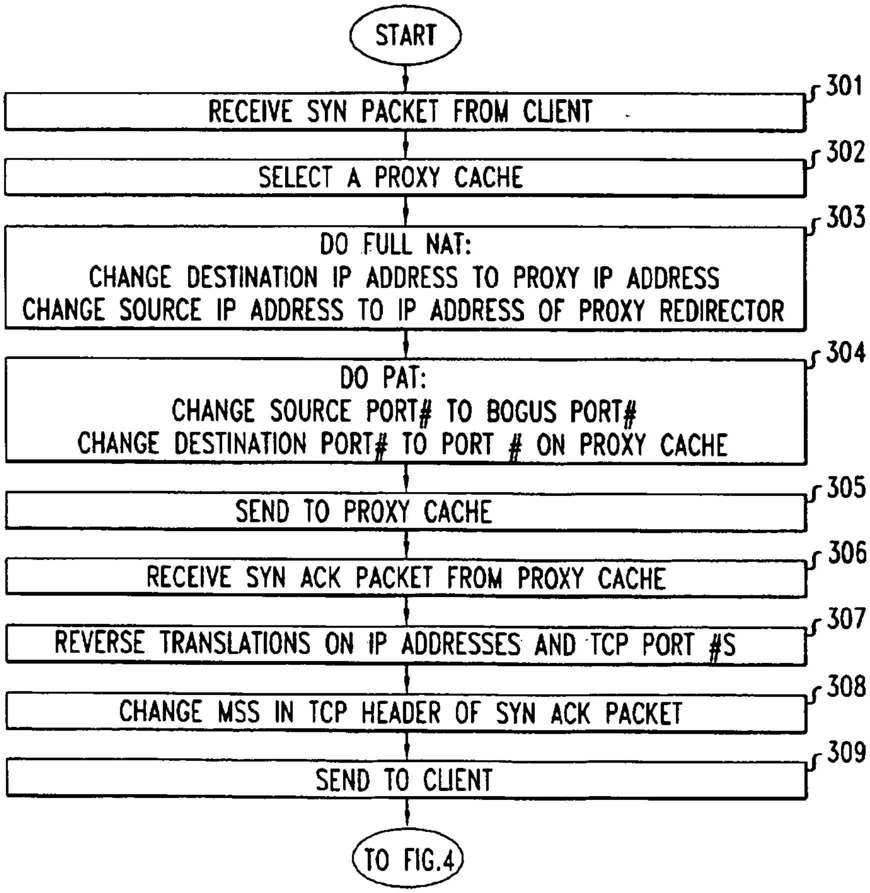

START
RECEIVE SYN PACKET FROM CLIENT
301
SELECT A PROXY CACHE
302
DO FULL NAT:
CHANGE DESTINATION IP ADDRESS TO PROXY IP ADDRESS
CHANGE SOURCE IP ADDRESS TO IP ADDRESS OF PROXY REDIRECTOR
303
DO PAT:
CHANGE SOURCE PORT# TO BOGUS PORT#
CHANGE DESTINATION PORT# TO PORT # ON PROXY CACHE
304
SEND TO PROXY CACHE
305
RECEIVE SYN ACK PACKET FROM PROXY CACHE
306
REVERSE TRANSLATIONS ON IP ADDRESSES AND TCP PORT #S
307
CHANGE MSS IN TCP HEADER OF SYN ACK PACKET
308
SEND TO CLIENT
309
TO FIG.4

FIG. 2

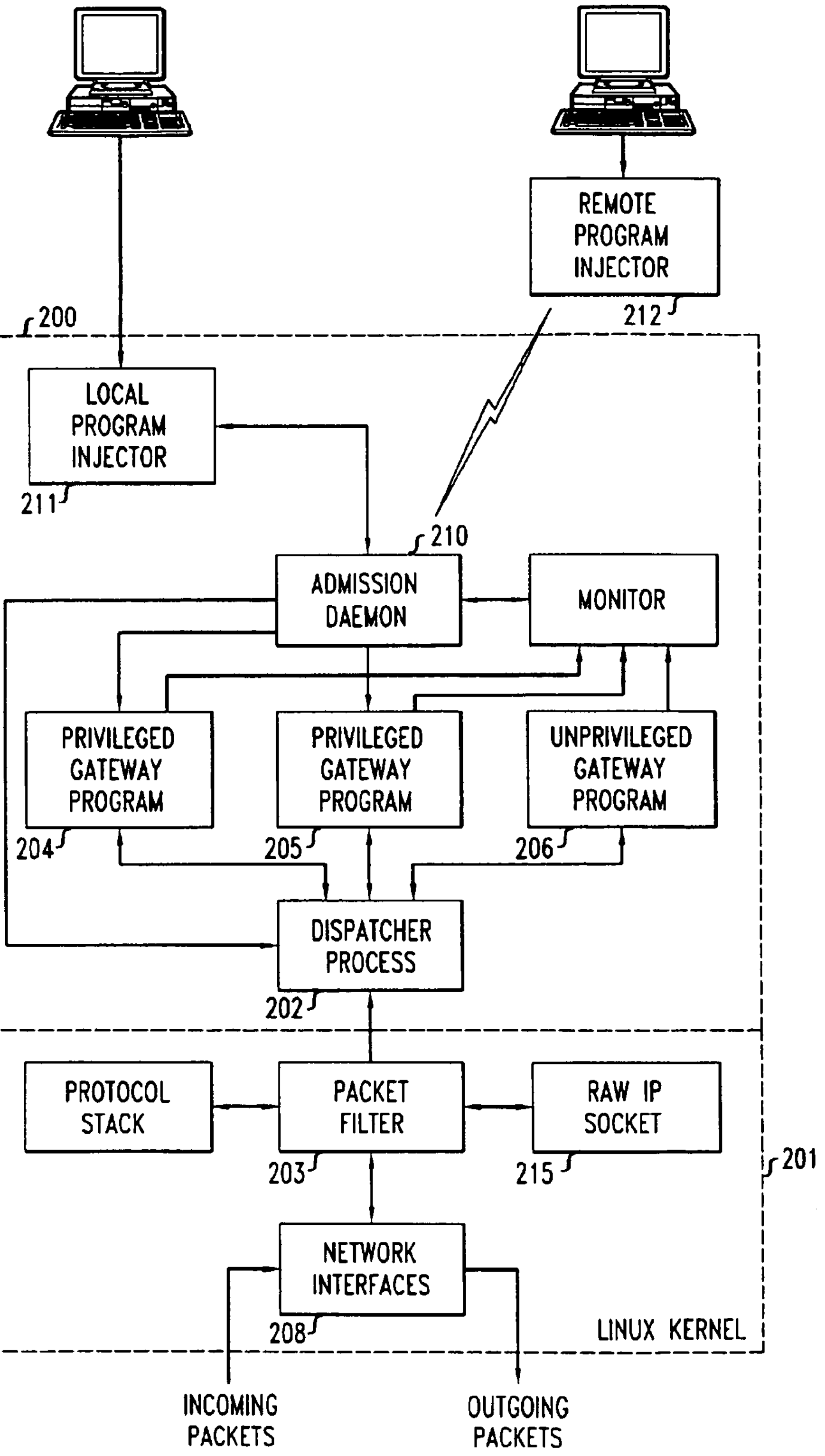

REMOTE PROGRAM INJECTOR
212
200
LOCAL PROGRAM INJECTOR
211
210
ADMISSION DAEMON
MONITOR
PRIVILEGED GATEWAY PROGRAM
PRIVILEGED GATEWAY PROGRAM
UNPRIVILEGED GATEWAY PROGRAM
204
205
206
DISPATCHER PROCESS
202
PROTOCOL STACK
PACKET FILTER
RAW IP SOCKET
203
215
201
NETWORK INTERFACES
208
LINUX KERNEL
INCOMING PACKETS
OUTGOING PACKETS

FIG. 4

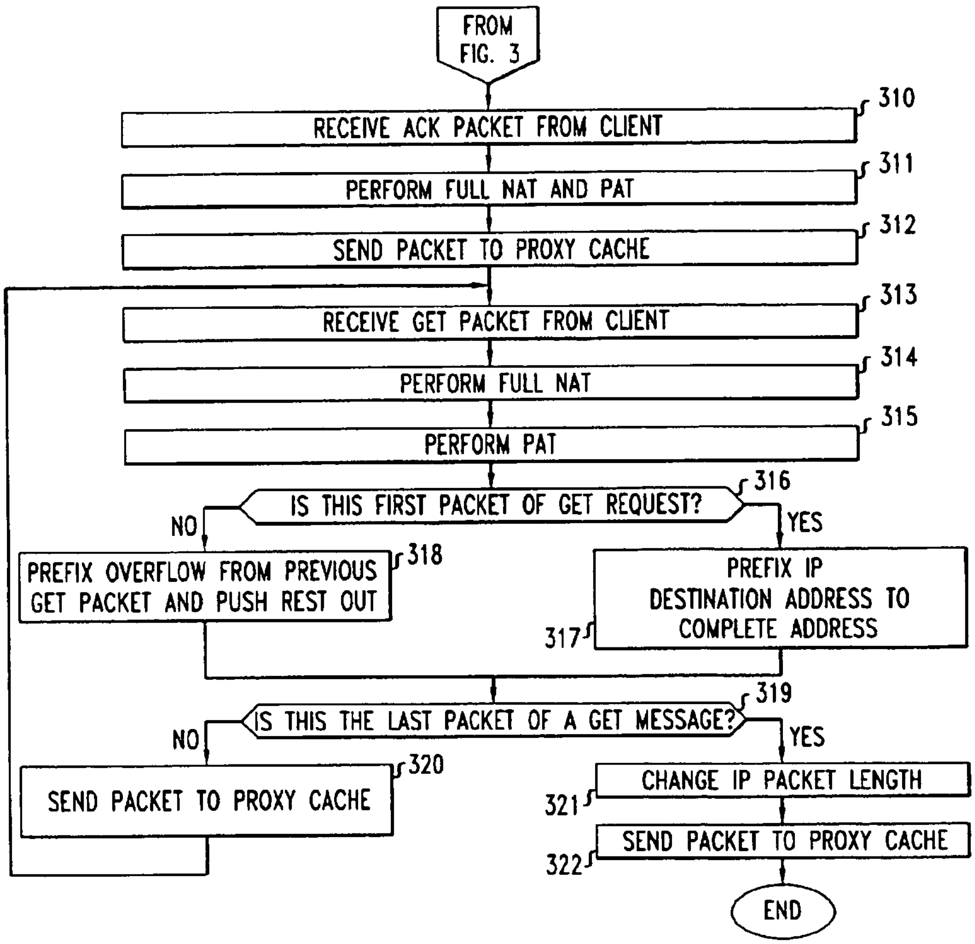

FROM FIG. 3
RECEIVE ACK PACKET FROM CLIENT
310
PERFORM FULL NAT AND PAT
311
SEND PACKET TO PROXY CACHE
312
RECEIVE GET PACKET FROM CLIENT
313
PERFORM FULL NAT
314
PERFORM PAT
315
IS THIS FIRST PACKET OF GET REQUEST?
316
NO
YES
PREFIX OVERFLOW FROM PREVIOUS GET PACKET AND PUSH REST OUT
318
PREFIX IP DESTINATION ADDRESS TO COMPLETE ADDRESS
317
IS THIS THE LAST PACKET OF A GET MESSAGE?
319
NO
YES
SEND PACKET TO PROXY CACHE
320
CHANGE IP PACKET LENGTH
321
SEND PACKET TO PROXY CACHE
322
END

FIG. 5

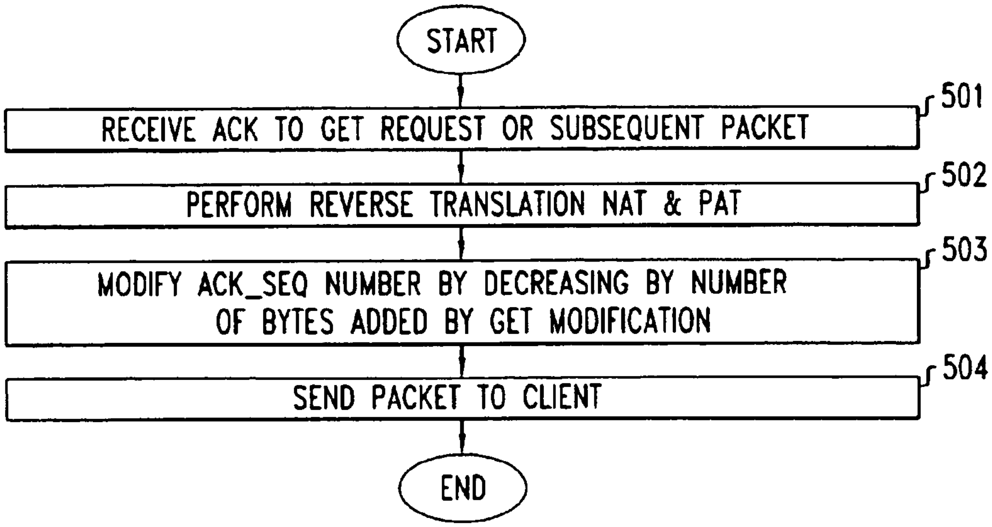
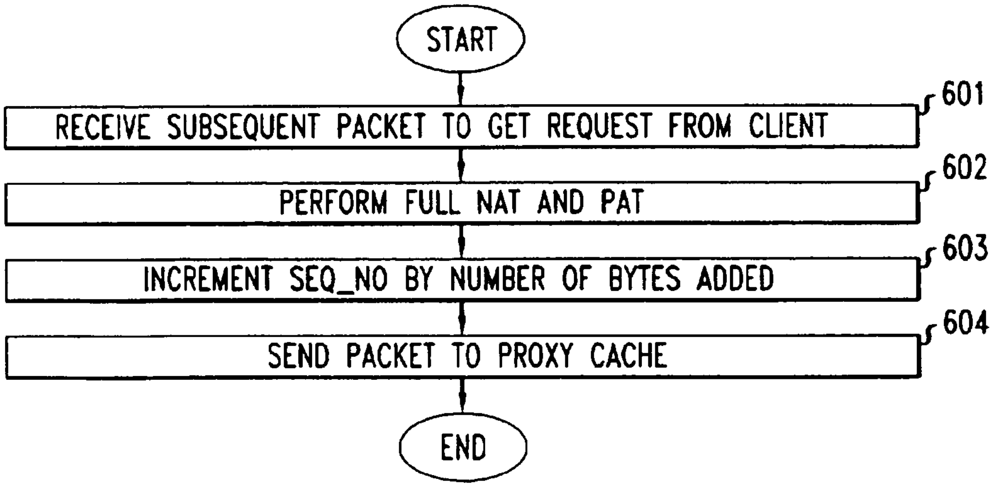

START
RECEIVE ACK TO GET REQUEST OR SUBSEQUENT PACKET
501
PERFORM REVERSE TRANSLATION NAT & PAT
502
MODIFY ACK_SEQ NUMBER BY DECREASING BY NUMBER OF BYTES ADDED BY GET MODIFICATION
503
SEND PACKET TO CLIENT
504
END

FIG. 6


START
RECEIVE SUBSEQUENT PACKET TO GET REQUEST FROM CLIENT
601
PERFORM FULL NAT AND PAT
602
INCREMENT SEQ_NO BY NUMBER OF BYTES ADDED
603
SEND PACKET TO PROXY CACHE
604
END

METHOD AND APPARATUS FOR TRANSPARENTLY DIRECTING REQUESTS FOR WEB OBJECTS TO PROXY CACHES

FIELD OF THE INVENTION

This invention relates to packet-switched computer networks, and more particularly, to a method and apparatus in such a network for transparently intercepting client web requests and redirecting them to proxy caches.

BACKGROUND OF THE INVENTION

Proxy caching is currently used to decrease both the latency of object retrieval and traffic on the Internet backbone. As is well known, if a proxy cache has stored a copy of an object from an origin server that has been requested by a client, the requested object is supplied to the client from the proxy cache rather than from the origin server. This, therefore, obviates the need to send the request over a wide area network, such as the Internet, to the origin server where the original object is stored and the responsive transmission of a copy of the requested object back over the network to the requesting client.

Direction of a request from a client to a proxy cache to determine whether a requested copy of an object is stored in the cache can be accomplished either transparently or nontransparently to the client. Non-transparent redirection is accomplished through the client's browser program which is configured to send all object requests to a designated proxy cache at a specified address. Generally, a browser can be configured to send all of its client requests to a designated proxy cache if the client is connected on a Local Area Network (LAN), or on an Intranet behind a firewall, where a proxy cache associated with that LAN or Intranet is located. When clients are served by a large Internet Service Provider (ISP), however, it is not advantageous from the ISP's standpoint to allow its subscribers to set their browsers to a specific proxy cache associated with the ISP. A large ISP likely will have many proxy caches in several locations and will thus want to maintain control over which of its several particular proxy caches a client request is directed. Further, if a proxy cache whose address is statically set in a client's browser becomes inoperative, all client requests will fail.

It is therefore more desirable from an ISP's standpoint with respect to latency and minimizing traffic onto and off of the network to transparently intercept a client's web request and send it to one of its operative proxy caches to determine whether a copy of the requested object is stored there. If a copy of the requested object is then found to be stored in that proxy cache, a copy of the object is sent to the client, which is unaware that it has been served an object from the proxy cache rather than from the origin server to which it made the request. If the proxy cache does not hold a copy of the requested object, then a separate connection is established between the proxy cache and the origin server to obtain a copy of the object, which when returned to the proxy is sent to the client over the connection established between the client and the proxy.

When a client specifies a URL of the object it is requesting a copy of, a Domain Name Server (DNS) look-up is performed to determine from the URL an IP address of an origin server which has that requested object. As a result of that look-up, an IP address is returned to the client of one of what may be several substantially equivalent servers that contain that object. The client then establishes a TCP connection to that server using a three-way handshake mechanism. Such a connection is determined at each end by a port number and an IP address. First, a SYN packet is sent from the client to that origin server, wherein the destination IP address specified in the packet is the DNS-determined IP address of the origin server and the destination port number for an HTTP request is conventionally port **80**. The source IP address and port number of the packet are the IP address and port number associated with the client. The client IP address is generally assigned to the client by an ISP and the client port number is dynamically assigned by the protocol stack in the client. The origin server then responds back to the client with an ACK SYN packet in which the destination IP address and destination port are the client's IP address and port number and the packet's source IP address and port number are the server's IP address and the server's port number, the latter generally being port **80**. After receipt of the ACK SYN packet, the client sends one or more packets to the origin server, which packets include a GET request. The GET request includes a complete URL, which identifies to that server the specific object within the origin server site that the client wants a copy of. Unlike an absolute URL, which includes both site information (e.g., www.yahoo.com), and object information (e.g., index.html), a complete URL only identifies the particular object (e.g., index.html) that is requested since the packet(s) containing the GET request is sent to the proper origin server site by means of the destination address of the packet(s).

When a browser is configured to non-transparently send all requests to a proxy, a GET request is formulated by the browser that includes the absolute URL of the requested object. That absolute URL is then used by the proxy to establish a separate TCP connection to the origin server if the proxy does not have a copy of the requested object in its cache. The proxy requires the absolute URL since the destination address of the packets to the proxy is set by the browser to the IP address of the proxy rather than the IP address of the origin server. Thus, in order to determine whether it has the object in its cache and if not establish a connection to the origin server, the proxy requires the absolute URL of the origin server in the GET request.

When requests are transparently directed to a proxy cache, however, the client browser is unaware that the request is being directed to the proxy and is possibly being fulfilled from the cache. Rather, the client's browser needs to "think" that it is connected to the origin server to which its SYN and the packet(s) containing the GET request are addressed. Such origin server IP address is determined by the browser through a. DNS look-up. Further, the source address of the ACK SYN packet and the packets containing the requested object must be that same origin server IP address or they will not be recognized by the browser as being the responsive packets to the SYN packet and the request for the object. Thus, in order to transparently send object requests to a proxy cache, a mechanism must be in place along the packet transmission path to intercept an initial SYN packet sent by a browser and to redirect it to the proxy cache to establish a TCP connection. The proxy cache must then masquerade as the origin server when sending the ACK SYN packet back to the client by using the origin server's IP address and port number as the source address of that packet. Further, the subsequent packet(s) containing a GET request must be redirected to the proxy cache and the request fulfilled either from the cache or via a separate TCP connection from the proxy to the origin server. In either case, the source address of packets sent back to the client must be the origin server's IP address and port number to which the packets sent by the client are addressed.

In order for packets associated with a request for an object to be redirected to a proxy cache connected somewhere in the network, a Layer **4** (L4) switch on the packet path "looks" at the port number of a destination address of a SYN request packet. Since HTTP connection requests are generally directed to port **80** of an origin server, the L4 switch transparently redirects all packets having a port number of **80** in the destination address. The SYN packet is thus sent to a selected proxy cache. In order for the proxy cache to properly respond to the client, as noted, it must know the absolute URL of the requested object and packets returned to the client must masquerade as coming from the origin server. Unlike the non-transparent caching method previously described in which the browser formulates a GET request with the absolute URL, for transparent caching the absolute URL must be provided in some manner to the proxy cache in order for the proxy to determine whether it in fact has the requested object in its cache, or whether it must establish a separate TCP connection to the origin server to request the object. In the prior art, when one or more caches are directly connected to the L4 switch, the switch chooses one of the caches and transparently forwards the packets to that proxy without modifying the source or destination address of the packets. The proxy, working in a promiscuous TCP mode accepts all incoming packets regardless of their destination address. The proxy, then receiving the SYN packet with the origin server's destination address and the client's source address, can respond to SYN packet with an ACK SYN packet. This ACK SYN packet has the client's address as a destination address and a source address masquerading as the origin server address. This packet is transported through the L4 switch onto the network over the TCP connection back to the client. The subsequent packet(s) with the GET request from the client is redirected by the L4 switch to the directly connected proxy. Since the GET packet(s) only contains the complete URL, the proxy must formulate the absolute URL to determine whether its has the requested object in its cache or whether is must establish a separate TCP connection to the origin server. The proxy forms the absolute URL by prefixing the complete URL in the GET request with the IP address of the origin server in the destination address of the packet. The proxy can then determine whether it has the object and, if not, establish a TCP connection to that absolute address. If that particular origin server at that IP address should be inoperative, the proxy can alternatively prefix the complete URL in the GET request with the logical name of the site indicated in the HOST field in the packet(s) containing the GET request.

In the prior art, if the proxy cache is not directly connected to the L4 switch, then the L4 switch must perform a network address translation (NAT) and port address translation (PAT) on those packets directed to port **80** of an origin server. Specifically, when the L4 switch receives a SYN packet to initiate a TCP connection from a client to an origin server, it translates the destination address of the packet from the IP address and port number of the origin server to the IP address and port number of a selected proxy cache. Further, the switch translates the source address of the packet from the clients IP address and port number to its own IP address and a port number. When the proxy responds with an ACK SYN packet, it therefore responds to the L4 switch where a NAT translates the destination IP address from the IP address of the L4 switch to the IP address of the client, and translates the source IP address from the IP address of the proxy to IP address of the origin server. A PAT also translates the port number in the destination address from that of the L4 switch to that of the client, and translates the port number in the source address from that of the proxy to that of the origin server (usually **80**). When the client sends an ACK packet and then the packet(s) containing the GET request to the origin server, the L4 switch again performs a NAT, translating the destination IP address to the IP address of the proxy. Thus, when the packet(s) containing the GET request is received by the proxy, it does not know the IP address of the origin server as in the directly connected proxy arrangement described above. The proxy must therefore look at the logical name in the HOST field and perform a DNS look-up to determine that site's IP address. The proxy then uses that IP address in combination with the complete URL in the GET request to form an absolute URL from which it determines whether it has the requested object in its cache. If it doesn't, a separate TCP connection is established from the proxy to that absolute URL to retrieve that object, which is returned to the proxy. Whether the object is found in the proxy cache or is retrieved over the separate connection from the origin server, it is forwarded back to the L4 switch where a NAT and PAT are performed to translate the destination address to that of the client and to translate the source address to the particular origin server to which the client's request was directed. It should be noted that the source address of the origin server obtained when the client's browser initiates a DNS look-up using the origin server's absolute URL may not be the same IP address obtained when the proxy performs a DNS look-up using the combination of the site URL in the HOST field and the complete URL in the GET request.

The above described techniques for performing transparent proxy caching have several disadvantages. Firstly, use of a HOST field to specify a logical name of an origin server is not currently incorporated within the presently employed HTTP1.0 standards. Thus, a HOST field may not be present in the packet(s) containing a GET request. Where, as described above, the information in the HOST field is necessary to form an absolute URL to determine whether the proxy cache has the requested object and, if not, to establish a connection to an origin server from the proxy, the absence of the HOST field results in an unfilled request. Secondly, the prior art techniques require the proxy cache to perform the function of forming an absolute URL from the information in the HOST field and in the packet(s) containing the GET request. Thus, standard proxy caches which expect the client's browser to produce the absolute URL cannot be used. A methodology for transparent proxy caching that is transparent to both the client and the proxy is desirable to avoid modification to the program that controls proxy cache operations. Standard proxy caches could thus be employed anywhere in the network without the need for a special implementation.

The above described prior art techniques have even further disadvantages with respect to persistent connections defined by the HTTP1.1 standards. As defined by these standards, a persistent connection enables a client to send plural GET requests over the same TCP connection once that connection has been established between two endpoints. When a prior art transparent proxy cache is interposed on the connection, a client may "think" it has established a persistent connection to the specific origin server determined through the DNS look-up. The connection in reality, however, is transparently diverted by the L4 switch to a proxy cache. The proxy cache, in response to a DNS look-up using the logical name in the HOST field, may be directed to an equivalent origin server at a different IP address. Further, as each subsequent GET request is received by the proxy from the client within the client's perceived persistent connection, each responsive DNS look-up to the logical name may direct a connection to an even different IP address of an equivalent origin server. As a result, the advantages of a transaction-oriented persistent connection in which a server is capable of maintaining state information throughout the connection, are lost. A methodology is desirable that maintains persistence to the same origin server to which the clients browser is directed, or to a same equivalent origin server throughout the duration of the persistent connection.

SUMMARY OF THE INVENTION

The problems associated with the prior art techniques for transparent proxy caching are eliminated by the present invention. In accordance with the present invention, a switching entity, such as the L4 switch (referred to hereinafter as a proxy redirector), through which the packets flow, is provided with the functionalities at the IP level necessary to transform the complete URL in each GET request transmitted by a client to an appropriate absolute URL. Specifically, the IP address found in the destination field in the IP header of the packet(s) from the client containing the GET request are added as a prefix by the proxy redirector to the complete URL in the GET request. As a result, the complete URL in the GET request is modified to form an absolute URL which, when received by the proxy cache, is directly used to determine if the requested object is stored in the cache and, if not, to establish a separate TCP connection to the origin server. The GET request received by the proxy is thus equivalent to what it would expect to receive if it were operating in the non-transparent mode. Advantageously, if a persistent connection is established, each subsequent GET request has the same IP address prefix determined by the initial DNS look-up by the client.

By modifying the GET request at the proxy redirector to include the destination address of the origin server, the number of bytes at the IP level in the packet containing the resultant absolute address are increased by the number of bytes in the prefix. Included in the header within each packet is a sequence number (seq) that provides an indication of the position of the first byte number in the payload. Thus, when the IP address is added to a packet, the sequence number of each of the subsequent packets needs to be incremented by the count of the added bytes. Further, an acknowledgement sequence number (ack_seq) in the header on the packets returned from the proxy or the origin server that logically follow receipt of the GET packet(s) at the origin server needs to be decremented by the proxy redirector before being forwarded to the client to avoid confusing the client with respect to what the sequence number of the next byte it sends should be. Further, if the GET request sent by the client encompasses more than one TCP segment, then the extra bytes in the first of the segments caused by the additional bytes added to the URL are shifted into the second segment, and the resultant now extra bytes in the second segment are shifted into the third segment, etc., until the last of the segments. In order to preclude the necessity of requiring an extra segment to be added to the GET request to accommodate the extra bytes, the client sending the GET request, is deceived into sending segments whose maximum size is less than what can actually be received by the proxy as indicated by a maximum segment size (MSS) field in packets from the proxy. The proxy redirector, upon receipt of the On, ACK SYN packet from the proxy, reduces the MSS parameter received from the proxy by the amount of the number of bytes that will be added to the GET request before that parameter is forwarded to the client. Thus, when the client next sends a GET request, each segment is limited to the reduced MSS, thereby insuring that the segment size of a last segment in a GET request after the IP address is prefixed by the proxy redirector to form the absolute URL (whether the GET request is one or more segments long) is less than or equal to the actual MSS that the proxy can receive.

BRIEF DESCRIPTION OF THE DRAWING

FIG. 1 is a block diagram of a network that includes a proxy redirector that transparently sends requests from a client to a proxy cache by changing the destination address of packets in a client request from that of the origin server to that of a proxy and the source address from that of the client to that of the switching entity and, in accordance with the present invention, modifies a GET request to include the destination address of the origin server;

FIG. 2 is a block diagram showing the proxy redirector implemented on a programmable network element that manipulates packets in accordance with instructions provided by a loaded program; and

FIGS. 3, 4, 5 and 6 are flow charts detailing the operation of the proxy redirector.

DETAILED DESCRIPTION

With reference to FIG. 1, a plurality of clients **101-1–101-N** are connected to a local area network (LAN) **102**, such as an Ethernet. LAN **102**, which, in turn, is connected through a router **103** to a Level **4** (L4) switch **104** (proxy redirector) which interfaces the LAN with a wide area network (WAN) **105**, such as the Internet. Although shown as two separate elements, the functionalities of router **103** and proxy redirector **104** can in actual practice be combined in a single unit. All requests from any of the clients connected to LAN **102** for objects stored in servers connected to the Internet **105** traverse proxy redirector **104** onto the Internet. The packets comprising such requests, which include the standardized packets that establish a TCP connection, are directed to an IP destination address and port number indicated in the IP header of each packet originating from a client source address that includes a client IP address and port number. Similarly, responses to such requests from an origin server connected to Internet **105** are directed via an IP destination address that is equal to the client's IP address and port number from which the request originated, and have as a source address the server's IP address and port number. All such packets directed to any of the clients **101-1–101-N** from any server connected to Internet **105** pass through proxy redirector **104**.

When any of the clients connected to LAN **102**, such as client **101-1**, makes a request through a browser for an object by specifying a logical URL, a domain name server (DNS) **106** connected locally or on Internet **105**, as shown, is accessed to perform a database look-up based on that logical name. An associated IP address is then returned to the browser. The IP address returned to the browser is the IP address of a particular origin server which contains the **5** object requested through the logical URL. Since a logical name may in fact be associated with a plurality of essentially equivalent origin servers, such as servers **107** and **109**, the particular IP address returned to the client browser chosen by DNS **106** may be determined in a round-robin manner. When DNS **106** selects an origin server corresponding to the logical URL, the IP address of the selected origin server, such as, for example, the IP address of origin server **107**, is returned to the browser in the requesting client **101-1**. That IP address then serves as the IP address to which packets directed to the origin server from the client are directed. Conventionally, http requests are usually directed to port **80** of an origin server.

With the IP address of the origin server determined and returned to the client, the browser establishes a TCP connection between the client and the origin server through a three-way handshaking process. Specifically, a SYN packet, addressed to the IP address of the selected origin server, is sent by the client. Handshaking is completed when the client receives an acknowledgement of receipt of that SYN packet through an ACK SYN packet sent by that origin server, and responds with a ACK packet to the origin server. The browser then sends a GET request that specifies the particular requested object.

In accordance with the present invention, once the IP address of the origin server corresponding to the logical URL name is determined through the DNS look-up, proxy redirector 104, rather than establishing a TCP connection to the origin server at the determined IP address, transparently establishes a TCP connection between the client and a proxy. If the requested object is stored in the cache, a copy of that object is transparently returned to the requesting client. A TCP connection, therefore, is not established over the Internet 105, to the actual origin server 107 to provide the requested object to the requesting client. The coss of transmitting the request to the origin server over the Internet and transmitting the copy of the requested object back over the Internet are thereby saved in addition to the time required for transmitting such a request over the Internet and waiting for a response from the origin server. If the proxy cache to which the request is directed does not contain the requested object, a separate TCP connection is established between the proxy cache and the origin server to obtain a copy of the requested object. When the proxy cache then receives the copy of the requested object from an origin server over that separate TCP connection, the copy is forwarded to the client over the original TCP connection that was established between the client and the proxy cache.

In the embodiment shown in FIG. 1, a proxy cache 110-1 is illustratively shown connected to a LAN 111, which is connected to LAN 102 through a router 112. Another proxy cache 115 is shown connected on a different LAN 116 through router 103. Other proxy caches can be located anywhere on LANs 102, 111, or 116, on another LAN connected to the Internet 105 such as proxy cache 117. Proxy redirector 104 selects one of the available proxy caches to which to forward client requests based on a metric such as least-loaded or round-robin, based on IP header information such as the origin server IP address. With respect to the latter, all objects from a specific origin server will be served by a specific proxy.

In the preferred embodiment described herein, proxy redirector 104 includes a programmable network element of the type described in co-pending U.S. patent application Ser. No. 09/190,355, filed Nov. 12, 1998, which application is incorporated herein by reference. As described in that application, that programmable network element in the preferred embodiment runs on a Linux machine. As shown in FIG. 2, the programmable network element 200 includes a dispatcher process 202 with which plural different gateway programs (204, 205 and 206) register and request access to IP packets that fit specific descriptions. Such programs are loaded through an admission daemon 210 via a local program injector 211 or a remote program injector 212. A gateway program, for example, can request access to incoming packets to network interface 208 that match certain source and destination IP address ranges and port numbers. The dispatcher process 202 uses a packet filter 203 in the Linux kernel 201 to obtain packets requested by the gateway programs and uses a raw IP socket 215 to send packets that have been manipulated in accordance with the gateway program back to the kernel for output back to the network through filter 203 through network interfaces 208. Library functions are provided in the programmable network element that enable a gateway program to communicate with the dispatcher process 202 to register rules that specify the type of IP packets that a gateway program wants diverted to it. A gateway program can request either a complete IP packet or only the IP and TCP header of a packet and can change both the header and payload of a packet.

In the present invention, that programmable network element is operative in combination with a gateway program that manipulates the destination and source addresses of packets flowing there through in a manner to be described, as well as modifying, as will be described, information in the packet(s) containing the GET request that specifies the URL of the requested object. Specifically, the programmable network element in combination with the gateway program operates on packets associated with HTTP requests, which are determined from the destination port number. As previously noted, HTTP requests are conventionally addressed to port 80 of an origin server. Thus, the programmable network element/gateway program which together comprise proxy redirector 104 in this embodiment, captures through the dispatcher process of the programmable network element, packets directed to port 80 and then performs address translations on those captured packets to readdress these packets to a selected proxy. With respect to address translations, the gateway program translates the destination IP address of packets addressed to the origin server to the IP address of a selected proxy cache and translates the source IP address of such packets from that of the client to the IP address of proxy redirector 104. Further, in order for proxy redirector 104 to identify requests from plural client terminals that are directed to the same proxy, the source port number is translated to a bogus ghost port number at the proxy redirector. Thus, when proxy cache responds, the packets transmitted by the cache have a destination IP address of proxy redirector 104 at that bogus port number, which is distinctly associated with the client. The gateway program within proxy redirector 104 then translates the IP destination address of these responsive packets from the proxy to the IP address of the client and translates the bogus destination port number to the port number from which the client originated its request. Further, the gateway program translates the source IP address of such responsive packets from that of the proxy to the IP address of the origin server and the port number to the port (80) to which the client's requests were originally directed. Thus, the packets which are returned to the client from the proxy masquerade as if they had originated from the origin server to which the client "believed" its request had been sent.

By performing the above-described network address translations (NATs) and port address translations (PATs), packets from a client 101-1 are transparently directed by proxy redirector 104 to a proxy cache. Responsive packets from the proxy cache are sent to proxy redirector 104 where they are redirected to client 101-1.

In establishing a TCP connection that is directed to an origin server, client 101-1 first transmits a SYN packet, which is intercepted by proxy redirector 104. Proxy redirector 104 selects a proxy cache, such as proxy 110-1, to redirect this request and creates a connection control block (CCB) to maintain information about the connection. Selection of the particular proxy is determined, as described above, by one of several possible algorithms. The CCB is used to store the client IP address and TCP port number and

Client
redirector
proxycache
destination

the origin server IP address and TCP port number, both of which are contained in the IP header of the SYN packet, and the chosen proxy's IP address. The destination address is then changed to that of the chosen proxy and the packet is sent back to the network for redirection to its new destination address of the proxy **110-1**. All subsequent packets that originate from the same client with the same TCP port number are then forwarded to the same proxy. Proxy **110-1** responds with an ACK SYN packet which is directed via its destination address to proxy redirector **110-1**. Proxy redirector **104** then translates the source IP address and port number to those of the origin server and the destination IP address and port number to those of the client. When the packet arrives at the client the client believes that it is connected to the origin server. The client then responds with an ACK packet to the origin server, which is redirected by proxy redirector **104** to proxy cache **110-1**, to complete the handshaking process.

After the TCP connection is established between client **101-1** and proxy cache **110-1**, client **101-1** sends one or more packets containing a GET request addressed to the origin server. Such packets are thus "captured" by proxy redirector **104** and redirected to proxy cache **110-1**. As previously discussed, the GET request sent by the client contains only the complete URL sent by the client browser which in itself provides insufficient information for the proxy cache to determine whether it has the requested object and, if not, to forward it to the origin serve,r which does. In accordance with the present invention, the gateway program that is operative with the programmable network element of the proxy redirector **104**, captures this packet or packets and, in addition to previously described address translations, transforms the complete URL to an absolute URL by prefixing it with the IP address of the origin server obtained from the destination IP address of the packet(s) containing the GET request. Thus, Level **7** (application) information is modified to assist in level **4** routing.

In order to make the URL transformation transparent to both the client and proxy cache endpoints, changes in IP and TCP headers are also required. Since the GET request modification increases the length of the IP packet that carries the GET request, the total length field on the IP header of this packet is increased by an offset. The offset amount is recorded in the CCB. In addition, the TCP header contains sequence numbers (seq) and acknowledgement sequence numbers (ack_seq) that need to be translated. The seq in the TCP header indicates the byte number of the first byte on this packet going from the sender to the receiver over the TCP session and the ack_seq indicates the byte number of the next byte that the sender expects to receive from the receiver. For all packets after the GET packet(s) that go from the client to the proxy cache, the seq is increased by an offset equal to the lengthof(absolute URL)-lengthof(complete URL) so that the seq matches the byte number of the byte that the proxy cache expects to receive from the client. Similarly, on all packets starting with the acknowledgement to the GET packet that go from the proxy cache to the client through the proxy redirector, the ack_seq is decreased by the same offset so that the ack_seq matches the byte number of the byte that the proxy cache would expect the client to send in the next packet following the GET packet. By performing these changes in the header, the client and proxy cache endpoints remain unaware of the modification in the GET packets from the complete URL to the absolute URL.

Table 1 illustrates the URL and other header transformations performed

TABLE 1

```
Arriving packet:

--->   beginning of packet header dump <---
--->   IP header: version=4 hdr_len=5 TOS=0 pkt_len=346 id=60276
       frag_off=4000H TTL=64 protocol=6 cksum=1792H
       saddr=135.104.25.243 daddr=204.71.200.244
--->   TCP header: sport=1273 dport=80 seq=2189084427 ack_seq=3266449517
       tcp_hdr_len=5 flags=ACK PSH
       res1=0H res2=0H window=31856 cksum=162H urgent=0
--->   beginning of packet data dump <---
GET /a/ya/yahoomail/promo1.gif HTTP/1.0
Referer: http://www.yahoo.com/
Connection: Keep-Alive
User-Agent: Mozilla/4.05 [en] (X11; U; Linux 2.1.103 i686)
Host: us.yimg.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg image/png
Accept-Language: en
Accept-Charset: iso-8859-1, *,utf 8
Modified to:

--->   beginning of packet header dump <---
--->   IP header: version=4 hdr_len=5 TOS=0 pkt_len=367 id=60276
       frag_off=4000H TTL=64 protocol=6 cksum=1792H
       saddr=135.104.25.245 daddr=135.104.25.31
--->   TCP header: sport=5000 dport=7000 seq=2189084427 ack_seq=3266449517
       tcp_hdr_len=5 flags=ACK PSH
       res1=0H res2=0H window=31856 cksum=162H urgent=0
--->   beginning of packet data dump <---
GET http://204.71.200.244/a/ya/yahoomail/promo1.gif HTTP/1.0
Referer: http://www.yahoo.com/
Connection: Keep-Alive
User-Agent: Mozilla/4.05 [en] (X11; U; Linux 2.1.103 i686)
Host: us.yimg.com
```

TABLE 1-continued

```
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg image/png
Accept-Language: en
Accept-Charset: iso-8859-1, *,utf 8
```

on a GET packet that arrives at proxy redirector **104** from a client **101-1**. The packet is destined to an origin server at a destination IP address (daddr) 204.71.200.244 (www.yahoo.com) at a destination port (dport) **80**, requesting object /a/ya/yahoomail/promo1.gif. As can be noted in the modified packet, the packet is redirected to a proxy cache **110-1** at IP address 135.104.25.31 port 7000 by changing the daddr and dport header information. Also, the complete URL of the object in the GET request is modified in the translated packet by prefixing it with http://204.71.200.244 to form the absolute URL, where that prefix is obtained from the daddr header in the arriving packet. This transformation increases the length of the packet by 21 bytes so that the pkt_len field in the header is modified from 346 to 367 bytes. Further, the source port is modified to a bogus port number by changing sport to 5000.

Table 2 shows the translations performed by the proxy redirector **104** to an acknowledgment from proxy cache **110-1** to the GET request. The arriving packet is addressed (daddr) to the IP address of the proxy redirector at the bogus port (dport) 5000 used by the proxy redirector for this TCP connection. The source IP address (saddr) and the source port (sport) are those of the proxy cache to where the GET request was directed. The ack_seq field indicates the byte number of the first byte that is expected to be sent in the packet following the GET packet. In this example, ack_seq is equal to the sequence number (seq) 218908427 of the GET packet plus the length of the GET packet, which in this case per Table 1 is 367. Thus ack_seq of the arriving packet is 218908754. Since client **101-1** is unaware that a proxy redirector has increased the length of the GET packet by 21 bytes, proxy redirector **104** decreases the ack_seq field by the number of bytes added, 21. Further, proxy redirector **104** translates the destination IP address and port number to those of the client **101-1**, and the source IP address and port number to that of the origin server. The modified packet, when received by the client, thus appears to the client to have originated from the origin server and the ack_seq field indicates a byte number that the client would next expect to send having previously sent a packet of length 367 bytes. All packets that are subsequently sent through the proxy redirector **104** to client **101-1** from proxy cache **110-1** are similarly modified to decrement the ack_seq field by the number of bytes, 21, added to the GET packet.

TABLE 2

```
Arriving packet:

--->   beginning of packet header dump <---
--->   IP header: version=4 hdr_len=5 TOS=0 pkt_len=40 id=14559
       frag_off=4000H TTL=255 protocol=6 cksum=10eH
       saddr=135.104.25.31 daddr=135.104.25.245
--->   TCP header: sport=7000 dport=5000 seq=3266449517 ack_seq=2189084754
       tcp_hdr_len=5 flags=ACK
       res1=0H res2=0H window=8433 cksum=32H urgent=0
Modified to:

--->   beginning of packet header dump <---
--->   IP header: version=4 hdr_len=5 TOS=0 pkt_len=40 id=14559
       frag_off=4000H TTL=255 protocol=6 cksum=10eH
       saddr=204.71.200.244 daddr=135.104.25.243
--->   TCP header: sport=80 dport=1273 seq=3266449517 ack_seq=2189084733
       tcp_hdr_len=5 flags=ACK
       res1=0H res2=0H window=8433 cksum=32H urgent=0
```

Table 3 illustrates a next packet sent by client **101-1** to the origin server after the GET packet. In this packet the sequence number (seq) is equal to the modified ack_seq sent to the client, as per Table 2. The destination IP address and port number are those of the origin server and the source IP address and port number are those of the client. When received by proxy redirector **104**, the packet is modified to change the source IP address and port number to the IP address and bogus port number of the proxy redirector. The destination address IP and port number are translated to that of proxy cache **110-1**. The sequence number (seq) is increased by that same value of 21 to match the byte number that the proxy cache expects to receive based on the sequence number previously received in the GET packet and the length, **367**, of the GET packet

TABLE 3

```
Arriving packet:

--->   beginning of packet header dump <---
--->   IP header: version=4 hdr_len=5 TOS=0 pkt_len=40 id=60281
       frag_off=4000H TTL=64 protocol=6 cksum=18bfH
       saddr=135.104.25.243 daddr=204.71.200.244
```

TABLE 3-continued

```
--->   TCP header: sport=1273 dport=80 seq=2189084733 ack_seq=3266450977
       tcp_hdr_len=5 flags=ACK
       res1=0H res2=0H window=31856 cksum=d3f7H urgent=0
Modified to:

--->   beginning of packet header dump <---
--->   IP header: version=4 hdr_len=5 TOS=0 pkt_len=40 id=60281
       frag_off=4000H TTL=64 protocol=6 cksum=18bfH
       saddr=135.104.25.245 daddr=135.104.25.31
--->   TCP header: sport=5000 dport=7000 seq=2189084754 ack_seq=3266450977
       tcp_hdr_len=5 flags=ACK
       res1=0H res2=0H window=31856 cksum=d3f7H urgent=0
```

received. All subsequent packets directed to the origin server from client **101-1** are similarly modified before being directed to proxy cache **110-1**.

In the above-description, it has been assumed that the length of the GET request both before modification, and after the URL extension is less than the maximum TCP segment size. In fact, the length of the GET request may be longer than one TCP segment. If the length of the GET request carrying the complete URL occupies x number of TCP segment and, after it is modified to carry the absolute URL, it still also fits into that same x number of TCP segments, then the segment carrying the URL is modified and overflowing characters are pipelined from one segment to the next. Thus, the overflowing characters from a previous packet are prefixed to the start of the next packet, etc., until the last packet, which length is increased by the increased number of bytes due to the URL modification. Therefore, the packet length of only the last segment is modified to include the characters that have been shifted into that segment. The ack_seq parameter in packets from the proxy cache to the client is modified starting from the acknowledgment to the last GET packet.

If the modification of the URL to the absolute URL could cause the last TCP segment of the GET request to overflow to another segment, a new TCP segment would need to be constructed and injected by the proxy redirector. The proxy redirector would then need to have the capability to retransmit this segment if it was lost. Thus, the proxy redirector would need to have some TCP layer functionalities. In order to avoid adding higher level functionality to the proxy redirector, segment sizes are limited to less than what the proxy cache is actually capable of receiving. When the complete URL is transformed to an extended URL, the maximum increase in size is 22 bytes, equal to the maximum length of an IP address of 15 bytes plus 7 bytes from the prefix: http://. The client is deceived to send segments whose maximum size is 22 bytes less than what the protocol allows it to send. The TCP segment size sent by the client is determined by what the proxy cache, in its handshake with the client, indicates as the maximum segment size it can receive. This is indicates by the proxy cache through the maximum segment size (MSS) field in the ACK SYN packet. Accordingly, the proxy redirector **104** intercepts the ACK SYN packets and decreases the specified MSS amount by 22. For example, if the MSS specified by the proxy cache is 1460, it is modified to 1438 by the proxy redirector before being sent to the client. When the client next sends a GET request, the TCP segments are limited to 1438 bytes. In the worst case, when the client sends a GET request, 22 bytes will be added to the xth TCP segment that carries this request. The length of this xth TCP segment will still then be within the maximum length specified by the proxy cache. If the event that the proxy cache does not stipulate a maximum MSS in the ACK SYN packet, the default used by the client is 536 bytes. An MSS option is then added by the proxy redirector to inform the client that the maximum MSS expected by the other end of the TCP connection is 514 bytes.

As previously described, a NAT and PAT are performed by proxy redirector **104** on all packets addressed by client **101-1** to an origin server, and all packets addressed by proxy cache **110-1** to proxy redirector **104** for return to the client. Proxy redirector **104** thus performs a NAT and a PAT on these packets flowing in both direction. If proxy redirector **104** selects a proxy cache that is located in such a point on the network that packets from the proxy cache addressed directly to client **101-1** must pass through proxy redirector **104** due to the network configuration, then proxy redirector need only perform a half-NAT on the packets flowing through it. Specifically, if proxy redirector **104** selects a proxy cache such as proxy cache **117**, all packets addressed to client **101-1** must pass through proxy redirector **104**. Proxy redirector **104** thus only needs to transform the destination IP address and port number of packets from client **101-1** to the IP address and port number of proxy cache **117**, while maintaining the source IP address and port number as those of client **101-1**. The packets returned from proxy cache **117** will thus be addressed to the client's IP address and port number. When they pass through proxy redirector **104**, they are captured and the transformation of the source IP address and port to those of the origin server are the only address changes that need to be effected.

The problems of the prior art with respect to persistent connections is obviated in accordance with the present invention. As previously noted, during a persistent connection plural GET requests can be made by a client. In the prior art, as described, each GET request can result in a connection from a proxy cache to a different origin server if the proxy cache does not have the requested objects. The ability of a server to maintain the state of a client's connection throughout the duration of the connection is compromised if each GET request results in connections to multiple servers. In accordance with the present invention, once the IP address of the origin server is determined at the initial DNS lookup, that same IP address is used by the proxy redirector as a prefix to each complete URL in every GET request issued by the client throughout the duration of the persistent connection. Thus, assuming the proxy cache does not contain any of the requested objects, the proxy cache will establish a TCP connection to the same origin server in response to each GET request generated by the client. It should be noted that if plural client GET requests are forwarded by the proxy redirector to a proxy cache within a persistent TCP connection, ack_seq parameter in packets that flow through the proxy redirector from the proxy following each GET request must reflect the cumulative changes effected by translating the complete URL to the absolute URL in each of the preceding GET requests within the same TCP connection. Similarly, in all packets received by the proxy redirector from the client directed to the origin server within a persistent TCP connection, the seq parameter must reflect cumulative changes.

FIGS. 3, and 4, together are flow charts detailing the functions of proxy redirector 104 in establishing a TCP connection to a proxy cache and modifying the GET request so that such requests can be transparently forwarded to the proxy. At step 301, a SYN packet arrives from the client at the proxy redirector At step 302, proxy redirector selects a proxy cache based on a load balancing algorithm or on an arbitrary or random selection. At step 303, proxy redirector performs a full NAT, changing the daddr from that of the origin server to that of the selected proxy and saddr from that of the client to that of the proxy redirector. At step 304 a PAT is performed, changing sport to that of a bogus ghost port number and dport to the proxy's port number. At step 305, the SYN packet is sent to the proxy. In response to that SYN packet, the proxy responds, at step 306, with a SYN ACK packet containing an MSS parameter in the TCP header. At step 307, a reverse translation is performed on both the IP addresses and port numbers, changing saddr and sport to those of the origin server and daddr and dport to those of the client. At step 308, the MSS field is changed by reducing the value of the MSS received from the proxy by 22. At step 309, the ACK SYN packet is sent to the client. At step 310, proxy redirector receives a responsive ACK packet from the client. At step 311, a full NAT and PAT are performed on that packet and, at step 312, the modified packet is sent to the proxy, thereby completing the handshake sequence.

At step 313, a packet containing a GET request is received from the client. A full NAT is performed at step 314 and a PAT is performed at step 315. A determination is made at decision step 316 whether this is a first packet in the GET request. If yes, at step 317, the IP address of the origin server obtained Is from daddr of the arriving packet is prefixed to the complete URL in the GET request. If, at step 316, the packet is not a first packet in a GET request, then, at step 318, the overflow bytes from the previous GET packet are prefixed to those bytes in the current packet and if the total number of bytes in the resultant packet is than the actual MSS sent by the proxy, the overflow bytes greater are buffered for prefixing to the next packet. After alternative steps 316 or 318, at step 319, a determination is made whether the current packet is the last packet of a GET request. If not, at step 320, the current packet is sent to the proxy and the flow returns to step 313 to receive the next packet in the GET request from the client. If at step 319, the current packet is the last packet in a GET message, then, at step 321, the pkt_len parameter of that packet is changed to reflect the change in length of the packet. At step 322, the modified packet is sent to the proxy.

FIG. 5 illustrates the steps performed by the proxy redirector for each packet received from the proxy starting from the ACK to the GET request through the end of the connection. At step 501, the proxy redirector receives the ACK to the GET request, or any other packet that logically follows the ACK to the GET request. At step 502, reverse NAT and PATs are performed, translating daddr and dport to those of the client and saddr and sport to those of the origin server. At step 503, ack_seq is decreased by the amount added in the preceding GET request. At step 504, the modified packet is sent to the client.

FIG. 6 illustrates the step performed by the proxy redirector for each packet destined for the origin server from the client that follows the GET request. At step 601, a packet subsequent to the GET request is received from the client. At step 602, a full NAT and PAT are performed. At step 603, seq_no is increased by the amount of bytes added by modifying the previous GET request. At step 604, the packet is sent to the proxy.

In the discussion above of FIGS. 3, 4, 5 and 6, it has been assumed that the proxy cache is located in a position such that packets directed to the client will not automatically flow through the proxy redirector. Thus, all packets from the proxy are addressed to the proxy redirector. Therefore, for packets flowing from the client, and packets flowing from the proxy, the proxy redirector performs a full NAT and PAT. If however, as previously described, the proxy cache selected by the proxy redirector is located on the network so that all packets from proxy to the client automatically flow through the proxy redirector, then, in the steps shown in FIGS. 3, 4, 5 and 6, only a half NAT needs to be performed.

Although described in conjunction the programmable network element shown in FIG. 2, the proxy redirector of the present invention could be implemented through other means, using hardware, software, or a combination of both. As an example, a level 4 switch having a fixed program to perform the required packet manipulations required by the present invention could be used.

As described, the proxy cache returns requested objects to the address from which a request originated as indicated by the saddr and sport parameters in the IP header information, which is the address of the proxy redirector 104 when the proxy cache is not connected on the network so that all responses do not automatically pass through the proxy redirector. The interactions between the requesting client and the proxy cache are transparent to both the client and the proxy cache, since the client does not "know" that its request is being redirected to the proxy by the proxy redirector, and the proxy cache, when receiving a GET request with an absolute URL does not know that that absolute URL is not being formulated by the client's browser operating in a non-transparent mode. Advantageously, the proxy cache requires no software modifications and standard proxy caches, connected anywhere on the network can be used in conjunction with the proxy redirector. If, however, the proxy is modified, using a programmable network element as previously described, for example, the requested object retrieved by the proxy from its own cache or received from an origin server, can be sent directly back to the client, thereby obviating the need to send such packets back to the proxy redirector for address translations and redirection to the client. By performing only a half-NAT at the proxy redirector and leaving the client's saddr and sport as the source IP address and port number in the header of the SYN packet, GET request packet(s), and other packets forwarded by proxy redirector 104 from the client, the proxy cache can return packets responsive to the request directly to the client by substituting the origin server's IP address and port number as the source address for its own address. If the proxy redirector performs a full NAT and PAT, then another mechanism must be incorporated to provide the client address to the proxy cache, such as incorporating the client address information as part of an appendix to the absolute address in the modified GET request and stripping the appended client address information at the proxy before determining whether the requested object is stored in the cache or whether a connection to the origin server need be made. Advantageously, by sending the packets from the proxy cache directly back to the client, the delay encountered by transmitting such packets back to the proxy redirector for address translation and redirection is eliminated. Disadvantageously, the proxy cache must be modified to perform these functions, precluding use of standard available proxy caches.

Although described hereinabove in connection with GET requests, the present invention can equally be applied to redirection of any type of request message in which the token is, for example, GET, POST or HEAD, or any other type of token yet to be implemented and/or standardized.

The foregoing therefore merely illustrates the principles of the invention. It will thus be appreciated that those skilled in the art will be able to devise various arrangements which, although not explicitly described or shown herein, embody the principles of the invention and are included within its spirit and scope. Furthermore, all examples and conditional language recited hereinabove are principally intended expressly to be only for pedagogical purposes to aid the reader in understanding the principles of the invention and the concepts contributed by the inventors to furthering the art, and are to be construed as being without limitation to such specifically recited examples and conditions. Moreover, all statements hereinabove reciting principles, aspects, and embodiments of the invention, as well as specific examples thereof, are intended to encompass both structural and functional equivalents thereof. Additionally, it is intended that such equivalents include both currently known equivalents as well as equivalents developed in the future, i.e., any elements developed that perform the same function, regardless of structure.

Thus, for example, it will be appreciated by those skilled in the art that the block diagrams and flowcharts described hereinabove represent conceptual views of illustrative circuitry and processes embodying the principles of the invention. Similarly, it will be appreciated that any flowcharts, flow diagrams, state transition diagrams, pseudocode, and the like represent various processes which may be substantially represented in computer readable medium and so executed by a computer or processor, whether or not such a computer or processor is explicitly shown.

The functions of the various elements shown in the FIGS., including functional blocks labeled as "processors" may be provided through the use of dedicated hardware as well as hardware capable of executing software in association with appropriate software. When provided by a processor, the functions may be provided by a single dedicated processor, by a single shared processor, or by a plurality of individual processors, some of which may be shared. Moreover, explicit use of the term "processor" or "controller" should not be construed to refer exclusively to hardware capable of executing software, and may implicitly include, without limitation, digital signal processor (DSP) hardware, read-only memory (ROM) for storing software, random access memory (RAM), and non-volatile storage. Other hardware, conventional and/or custom, may also be included. Similarly, any switches shown in the FIGS. are conceptual only. Their function may be carried out through the operation of program logic, through dedicated logic, through the interaction of program control and dedicated logic, or even manually, the particular technique being selectable by the implementor as more specifically understood from the context.

In the claims hereof any element expressed as a means for performing a specified function is intended to encompass any way of performing that function including, for example, a) a combination of circuit elements which performs that function or b) software in any form, including, therefore, firmware, microcode or the like, combined with appropriate circuitry for executing that software to perform the function. The invention as defined by such claims resides in the fact that the functionalities provided by the various recited means are combined and brought together in the manner which the claims call for. Applicant thus regards any means which can provide those functionalities as equivalent to those shown hereinabove.

What is claimed is:

1. A method at a Layer 4 switch for redirecting an HTTP connection request from a client that is directed to an origin server to a proxy cache over a packet-based computer network comprising the steps of:

receiving at least one packet containing a request message within the HTTP connection, the at least one packet having an IP header comprising a destination address of an origin server, the request message including a complete address of a specified object at the origin server;

modifying, at the packet level, the request message by combining the destination address of the origin server with the complete address; and

forwarding the at least one packet containing the modified request message to the proxy cache over the packet-based network;

wherein the proxy cache is a standard proxy cache and, in the forwarding step, the at least one packet containing the modified request is redirected to the standard proxy cache transparently from the standpoints of both the client and the proxy cache, the proxy cache being able to determine whether it can satisfy the request for the specified object by using the combined destination address of the origin server and the complete address contained in the modified request message.

2. The method of claim 1 wherein the step of modifying comprises the step of prefixing the complete address in the request message with the destination address of the origin server.

3. The method of claim 2 further comprising the step of:

translating the destination address in an IP header of the at least one packet containing the modified request message to the address of the proxy cache.

4. The method of claim 3 further comprising the step of:

translating a source address in the IP header of the at least one packet containing the modified request message from an address of the client to an address of the Layer 4 switch.

5. The method of claim 3 further comprising the step of:

in a TCP header in packets received from the proxy cache commencing with an acknowledgment to the modified request message, modifying a value in an acknowledged byte sequence number field to reflect the increased number of bytes in the modified request message resulting from the step of prefixing the complete address with the destination address of the origin server.

6. The method of claim 5 wherein the step of modifying the value in the acknowledged byte sequence number field comprises the step of decreasing the value in the acknowledged byte sequence number field by the number of bytes added by prefixing the destination address of the origin server to the complete address in the request message.

7. The method of claim 6 further comprising the step of:

translating in the IP header the destination address to the address of the client and the source address to that of the origin server in the packets received from the proxy cache commencing with the acknowledgment to the request message.

8. The method of claim 3 further comprising the step of:

modifying a value in a sent byte sequence number field to reflect the increased number of bytes in the modified request message resulting from prefixing the complete address with the destination address of the origin server in a TCP header in packets received from the client following receipt of the request message.

9. The method of claim 8 wherein the step of modifying the value in the sent byte sequence number field comprises the step of increasing the value in the sent byte sequence number field by the number of bytes added by prefixing the destination address of the origin server to the complete address in the request message.

10. The method of claim 9 further comprising the step of:

translating the destination address to the address of the proxy cache in the IP header in each packet received from the client following receipt of the request message.

11. The method of claim 2 further comprising, prior to the step of receiving the at least one packet containing the request message, the steps of:

receiving a packet from the proxy cache indicating a maximum segment size that packets sent to it thereafter should have;

reducing by a predetermined number the maximum segment size of packets to be thereafter sent to the proxy cache; and

sending a packet to the client indicating the reduced maximum segment size that packets thereafter sent by the client to the origin server should have.

12. The method of claim 11 wherein the predetermined number is equal at least to the maximum number of bytes added by the step of prefixing the destination address to the complete address in the request message.

13. The method of claim 12 further comprising the steps of:

successively shifting to a next packet in the request message overflow bites caused by the step of prefixing the destination address to the complete address in the request message, if the request message is contained within more than one packet; and

changing a length-of-packet parameter in the IP header in a last packet of a multi-packet request message to reflect the bytes shifted into the last packet from a previous packet.

14. The method of claim 12 further comprising the step of changing a length-of-packet parameter in the IP header to reflect the change in the length of that packet caused by the step of prefixing the destination address to the complete address in the request message, if the request message is contained within one packet.

15. The method of claim 1 wherein the request message is a GET request.

16. The method of claim of claim 1 further comprising the step of selecting the proxy cache from a plurality of different proxy caches prior to receiving the at least one packet containing the request message.

17. The method of claim 1 wherein the request message is a POST request.

18. The method of claim 1 wherein the request message is a HEAD request.

19. A proxy redirector for redirecting an HTTP connection request from a client that is directed to an origin server to a proxy cache over a packet-based computer network comprising:

means for receiving at least one packet containing the request message, the a least one packet having an IP header comprising a destination address of an origin server, the request message including a complete address of a specified object at the origin server;

means for modified the request message by combining the destination address of the origin server with the complete address at the packet level; and

means for forwarding the at least one packet containing the modified request message to the proxy cache over the packet-based network;

wherein the proxy cache is a standard proxy cache and the means for forwarding redirects the at least one packet containing the modified request to the standard proxy cache transparently from the standpoints of both the client and the proxy cache, the proxy cache being able to determine whether it can satisfy the request for the specified object by using the combined destination address of the origin server and the complete address contained in the modified request message.

20. The proxy redirector of claim 19 wherein the means for modifying comprises means for prefixing the complete address in the request message with the destination address of the origin server.

21. The proxy redirector of claim 20 further comprising:

means for translating the destination address in an IP header of the at least one packet containing the modified request message to the address of the proxy cache.

22. The proxy redirector of claim 21 further comprising:

means for translating a source address in the IP header of the at least one packet containing the modified request message from an address of the client to an address of the proxy redirector.

23. The proxy redirector of claim 21 further comprising:

means for modifying a value in an acknowledged byte sequence number field in a TCP header in packets received from the proxy cache, commencing with an acknowledgment to the modified request message, to reflect the increased number of bytes in the modified request message resulting from prefixing the complete address with the destination address of the origin server.

24. The proxy redirector of claim 23 wherein the means for modifying the value in the acknowledged byte sequence number field decreases the value in the acknowledged byte sequence number field by the number of bytes added by prefixing the destination address of the origin server to the complete address in the request message.

25. The proxy redirector of claim 24 further comprising:

means for translating in the IP header the destination address to the address of the client and the source address to that of the origin server in the packets received from the proxy cache commencing with the acknowledgment to the request message.

26. The proxy redirector of claim 21 further comprising:

means for modifying a value in a sent byte sequence number field to reflect the increased number of bytes in the modified request message resulting from prefixing the complete address with the destination address of the origin server in a TCP header in packets received from the client following receipt of the request message.

27. The proxy redirector of claim 26 wherein the means for modifying the value in the sent byte sequence number field increases the value in the sent byte sequence number field by the number of bytes added by prefixing the destination address of the origin server to the complete address in the request message.

28. The proxy redirector of claim **27** further comprising:

means for translating the destination address to the address of the proxy cache in the IP header each packet received from the client following receipt of the request message.

29. The proxy redirector of claim **20** further comprising:

means for receiving a packet from the proxy cache indicating a maximum segment size that packets sent to it thereafter should have;

means for reducing by a predetermined number the maximum segment size of packets to be thereafter sent to the proxy cache; and

means for sending a packet to the client indicating the reduced maximum segment size that packets thereafter sent by the client to the origin server should have.

30. The proxy redirector of claim **29** wherein the predetermined number is equal at least to the maximum number of bytes added by the prefix of the destination address to the complete address in the request message.

31. The proxy redirector of claim **30** further comprising:

means for successively shifting to a next packet in the request message overflow bytes caused by the prefix of the destination address to the complete address in the request message, if the request message is contained within more than one packet; and

means for changing a length-of-packet parameter in the IP header in a last packet of a multi-packet request message to reflect the bytes shifted into the last packet from a previous packet.

32. The proxy redirector of claim **30** wherein the proxy cache further comprises means for changing a length-of-packet parameter in the IP header to reflect the change in the length of that packet caused by the prefix of the destination address to the complete address in the request message, if the length of the request message is contained within one packet.

33. The proxy redirector of claim **19** further comprising means for selecting the proxy cache from a plurality of different proxy caches.

34. The proxy redirector of claim **19** wherein the means for modifying the request message is a gateway program dynamically loaded on a programmable network element.

35. The proxy redirector of claim **19** wherein the request message is a GET request.

36. The proxy redirector of claim **19** wherein the request message is a POST request.

37. The proxy redirector of claim **19** wherein the request message is a HEAD request.

38. A computer readable medium storing computer program instructions which are executable on a computer system implementing a Layer **4** switch for redirecting an HTTP connection request from a client to a proxy cache over a packet-based computer network, said computer program instructions comprising instructions defining the steps of:

receiving at least one packet containing the request message, the at least one packet having an IP header comprising a destination address of an origin server, the request message including a complete address of a specified object at the origin server;

modifying, at the packet level, the request message by combining the destination address of the origin server with the complete address; and

forwarding the at least one packet containing the modified request message to the proxy cache over the packet-based network;

wherein the proxy cache is a standard proxy cache and, in the forwarding step, the at least one packet containing the modified request is redirected to the standard proxy cache transparently from the standpoints of both the client and the proxy cache, the proxy cache being able to determine whether it can satisfy the request for the specified object by using the combined destination address of the origin server and the complete address contained in the modified request message.

39. The computer readable memory of claim **38** wherein the step of modifying comprises the step of prefixing the complete address in the request message with the destination address of the origin server.

40. The computer readable memory of claim **39** wherein said computer program instructions further comprise instructions defining the step of:

translating the destination address in an IP header of the at least one packet containing the modified request message to the address of the proxy cache.

41. The computer readable memory of claim **40** wherein said computer program instructions further comprise instructions defining the step of:

translating a source address in the IP header of the at least one packet containing the modified request message from an address of the client to an address of the Layer **4** switch.

42. The computer readable memory of claim **40** wherein said computer program instructions further comprise instructions defining the step of:

in a TCP header in packets received from the proxy cache commencing with an acknowledgment to the modified request message, modifying a value in a acknowledged byte sequence number field to reflect the increased number of bytes in the modified request message resulting from the step of prefixing the complete address with the destination address of the origin server.

43. The computer readable memory of claim **42** wherein the step of modifying the value in the acknowledged byte sequence number field comprises the step of decreasing the value in the acknowledged byte sequence number field by the number of bytes added by prefixing the destination address of the origin server to the complete address in the request message.

44. The computer readable memory of claim **43** wherein said computer program instructions further comprise instructions defining the step of:

translating in the IP header the destination address to the address of the client and the source address to that of the origin server in the packets received from the proxy cache commencing with the acknowledgment to the request message.

45. The computer readable memory of claim **40** wherein said computer program instructions further comprise instructions defining the step of:

modifying a value in a sent byte sequence number field to reflect the increased number of bytes in the modified request message resulting from prefixing the complete address with the destination address of the origin server in a TCP header in packets received from the client following receipt of the request message.

46. The computer readable memory of claim **45** wherein the step of modifying the value in the sent byte sequence number field comprises the step of increasing the value in the sent byte sequence number field by the number of bytes added by prefixing the destination address of the origin server to the complete address in the fequest message.

47. The computer readable memory of claim **46** wherein said computer program instructions further comprise instructions defining the step of:

translating the destination address to the address of the proxy cache in the IP header each packet received from the client following receipt of the request message.

48. The computer readable memory of claim 39 wherein said computer program instructions further comprises instructions defining, prior to the step of receiving the at least one packet containing the request message, the steps of:

receiving a packet from the proxy cache indicating a maximum segment size that packets sent to it thereafter should have;

reducing by a predetermined number the maximum segment size of packets to be thereafter sent to the proxy cache; and

sending a packet to the client indicating the reduced maximum segment size that packets thereafter sent by the client to the origin server should have.

49. The computer readable memory of claim 48 wherein the predetermined number is equal at least to the maximum number of bytes added by the step of prefixing the destination address to the complete address in the request message.

50. The computer readable memory of claim 49 wherein said computer program instructions further comprise instructions defining the steps of:

successively shifting to a next packet in the request message overflow bytes caused by the step of prefixing the destination address to the complete address in the request message, if the request message is contained within more than one packet; and

changing a length-of-packet parameter in the IP header in a last packet of a multi-packet request message to reflect the bytes shifted into the last packet from a previous packet.

51. The computer readable memory of claim 49 wherein said computer program instructions further comprise instructions defining the step of changing a length-of-packet parameter in the IP header to reflect the change in the length of that packet caused by the step of prefixing the destination address to the complete address in the request message, if the request message is contained within one packet.

52. The computer readable memory of claim of claim 38 wherein said computer program instructions further comprise instructions defining the step of selecting the proxy cache from a plurality of different proxy caches prior to receiving the at least one packet containing the request message.

53. The computer readable memory of claim 38 wherein the request message is a GET request.

54. The computer readable memory of claim 38 wherein the request message is a POST request.

55. The computer readable memory of claim 38 wherein the request message is a HEAD request.

* * * * *

US006157644A

United States Patent [19]

Bernstein et al.

[11] **Patent Number: 6,157,644**

[45] **Date of Patent: Dec. 5, 2000**

[54] **METHOD AND APPARATUS FOR ACCELERATING OSI LAYER 3 ROUTERS**

[75] Inventors: **Gregory M. Bernstein**, Fremont, Calif.; **Alan Chapman**, Kanata, Canada; **Philip Edholm**, Fremont, Calif.; **Jeffrey T. Gullicksen**, San Jose, Calif.; **Kenneth Gullicksen**, Campbell, Calif.

[73] Assignee: **Northern Telecom Limited**, Montreal, Canada

[21] Appl. No.: **08/946,431**

[22] Filed: **Oct. 7, 1997**

[51] **Int. Cl.**⁷ **H04L 12/28**

[52] **U.S. Cl.** **370/392**; 370/389; 370/400; 370/401

[58] **Field of Search** 370/389, 410, 370/400, 401, 402, 392, 351, 356, 395

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,398,245	3/1995	Harriman, Jr.	370/389
5,546,390	8/1996	Stone	370/408
5,550,816	8/1996	Hardwick et al.	370/397
5,566,170	10/1996	Bakke et al.	370/392
5,598,410	1/1997	Stone	370/469
5,835,710	11/1998	Nagami et al.	370/351
5,920,566	7/1999	Hendel et al.	370/356
5,920,699	7/1999	Bare	370/395

OTHER PUBLICATIONS

RND, "F.I.R.S.T: Fast Intranet Routed Switching Technology," White Paper, www.rndnetworks.com, (Jan. 1997).

Comer, D., "Internetworking with TCP/IP,", vol. 1, Prentice–Hall, 3rd Ed. pp. 20–32, 89–107, 109–121, 123–133 (1995).

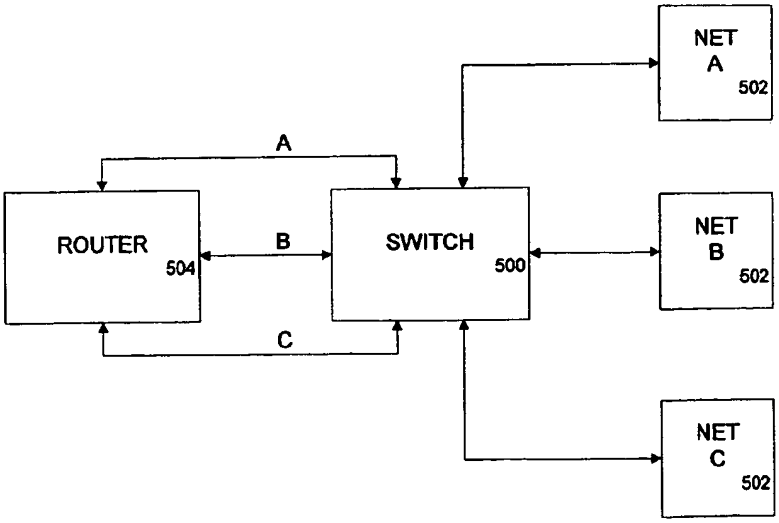
Rekhter, Y., "Inter–domain routing: EGP, BGP, and IDRP," in Steenstrup, M. *Routing in Communications Networks*, 1st Ed, pp. 99–133. (1995).

Primary Examiner—Chi H. Pham
Assistant Examiner—Brenda H. Pham
Attorney, Agent, or Firm—Morrison & Foerster LLP

[57] **ABSTRACT**

Methods and apparatus for accelerating a router in a communications network are described. In one embodiment, a router accelerator includes a forwarding table for associatively storing a destination address and a next hop address. If a destination address of a packet matches a destination address in the forwarding table, then logic forwards the packet to a next hop. A router may ~~be coupled to at least~~ one network port through the router accelerator. In another approach, the network includes at least one host and at least one router. The host has at least one routing table for associatively storing a second-level destination address, a second-level next hop address and a first-level next hop address. A router accelerator includes redirect logic for storing the second-level destination address in a second-level next hop address entry in the at least one host routing table. This causes the host to request a first-level next hop address corresponding to the second-level destination address. An accelerator table responds to the host's request with the bound first-level next hop address. In another approach employing host routing tables, a router accelerator includes request logic for requesting from the router a second-level next hop address in response to the first-level next hop address. Redirect logic stores the second-level next hop address in a second-level next hop address entry in the at least one host routing table. This causes the host to request a first-level next hop address corresponding to the second-level next hop address.

36 Claims, 11 Drawing Sheets



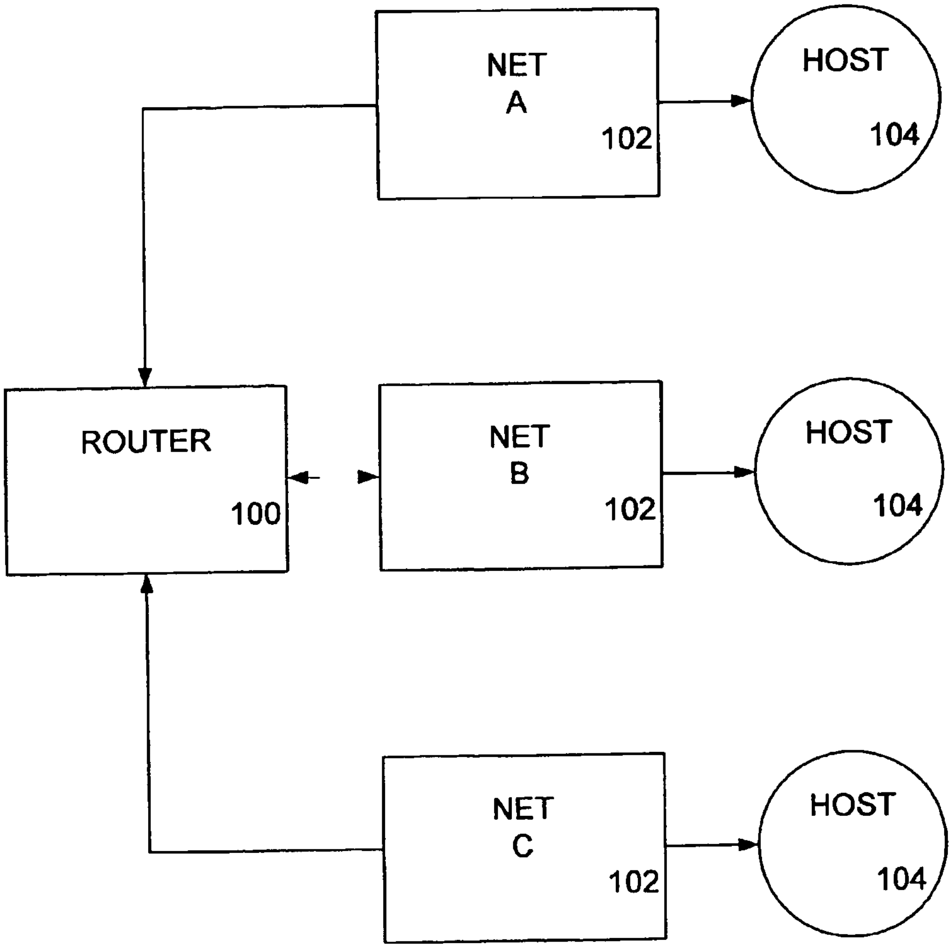

NET
A
102
HOST
104
ROUTER
100
NET
B
102
HOST
104
NET
C
102
HOST
104

Figure 1

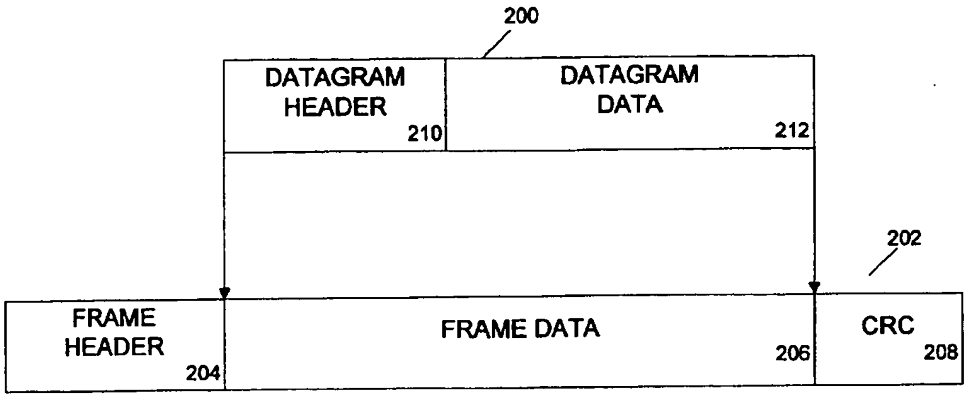


Figure 2

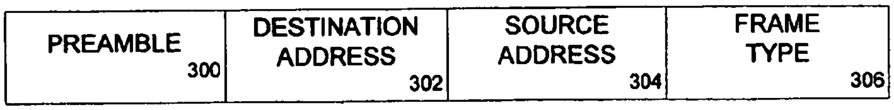


Figure 3

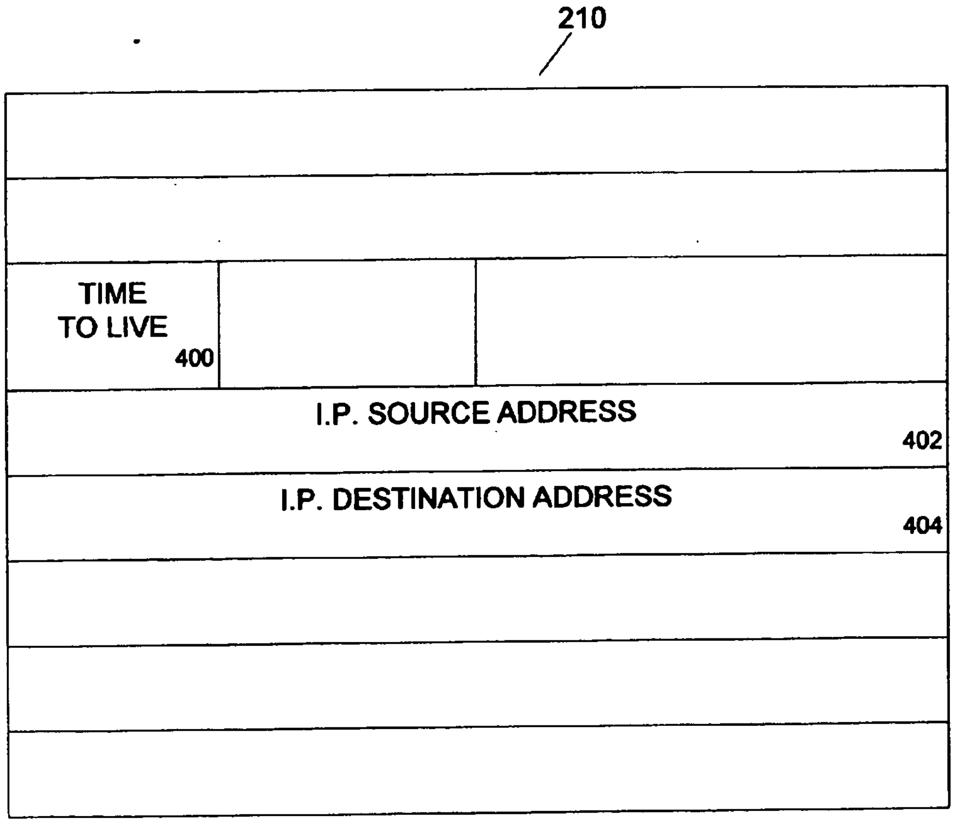

210
TIME TO LIVE
400
I.P. SOURCE ADDRESS
402
I.P. DESTINATION ADDRESS
404

Figure 4

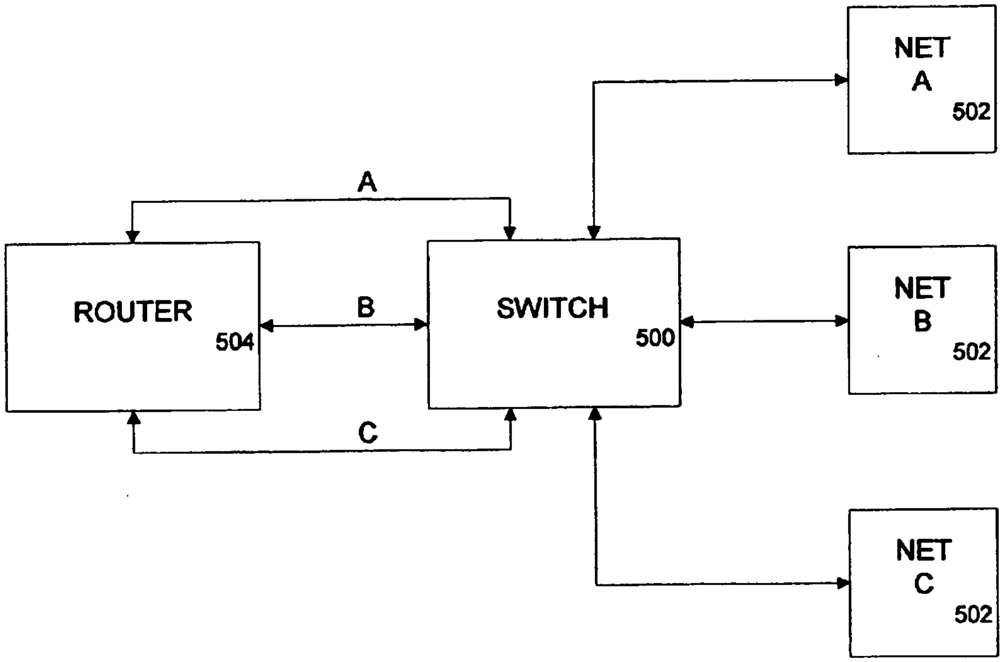


Figure 5

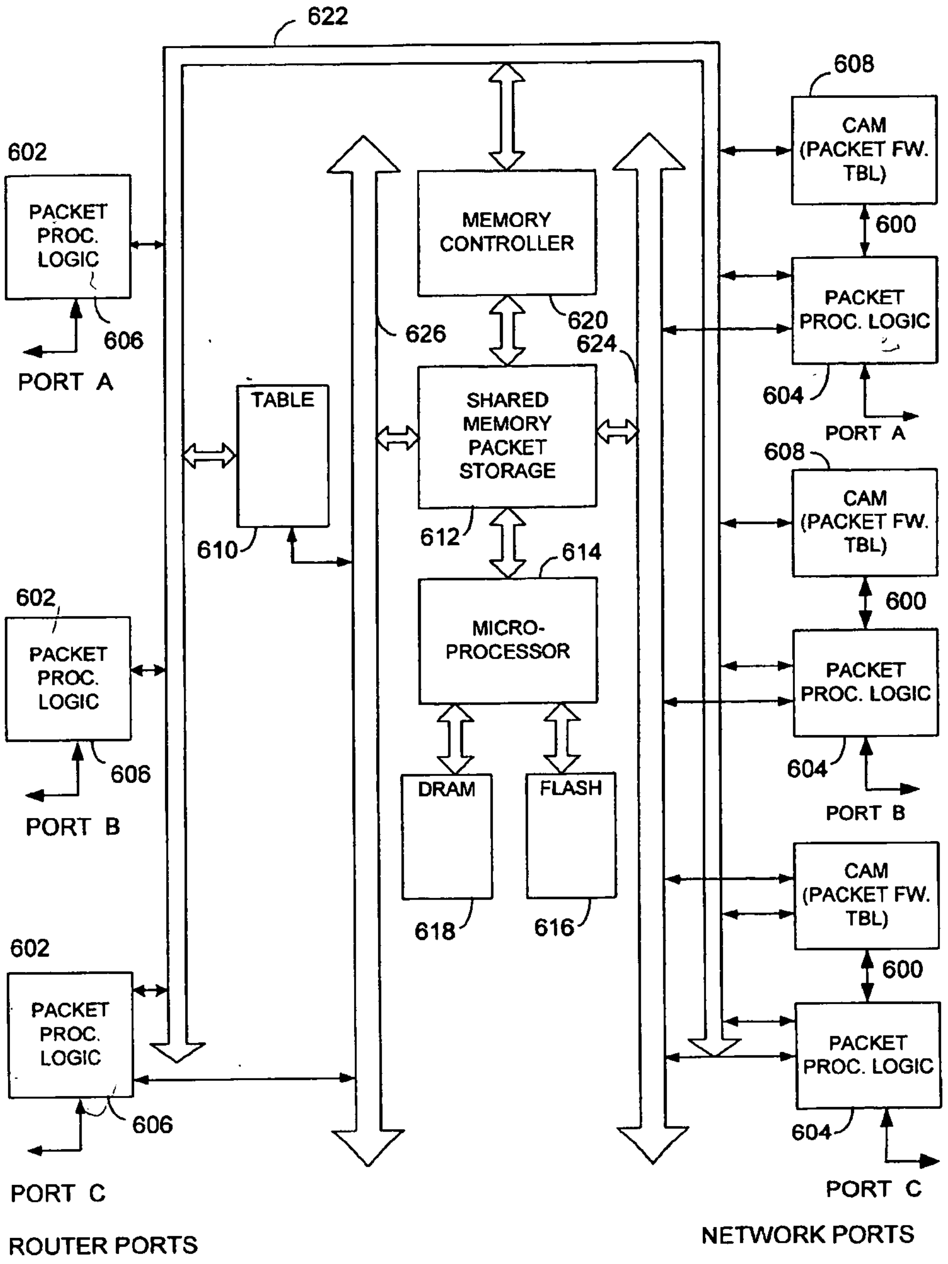

622
608
602
CAM
(PACKET FW.
TBL)
PACKET
PROC.
LOGIC
MEMORY
CONTROLLER
600
PACKET
PROC. LOGIC
606
626
620
624
PORT A
TABLE
SHARED
MEMORY
PACKET
STORAGE
604
PORT A
608
CAM
(PACKET FW.
TBL)
612
610
614
602
600
MICRO-
PROCESSOR
PACKET
PROC.
LOGIC
PACKET
PROC. LOGIC
606
604
DRAM
FLASH
PORT B
PORT B
CAM
(PACKET FW.
TBL)
618
616
602
600
PACKET
PROC.
LOGIC
PACKET
PROC. LOGIC
606
604
PORT C
PORT C
ROUTER PORTS
NETWORK PORTS

Figure 6

I.P. DESTINATION ADDRESS	LAYER 2 ADDRESS	DESTINATION PORT	TIME TO EXIST
⋮			

Figure 7

I.P. DESTINATION ADDRESS	INPUT PORT
⋮	

Figure 8

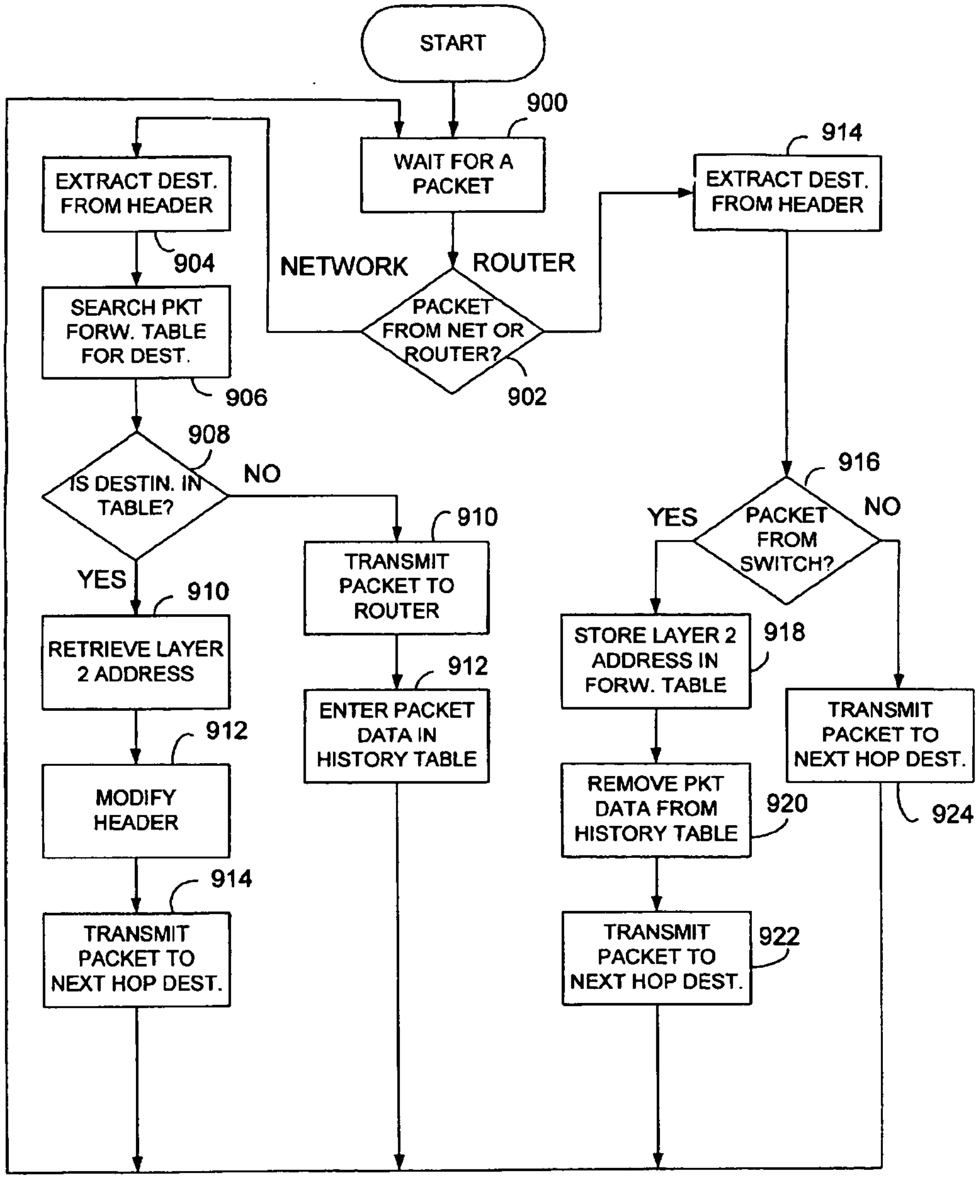

START
900
WAIT FOR A PACKET
NETWORK
ROUTER
PACKET FROM NET OR ROUTER?
902
EXTRACT DEST. FROM HEADER
904
SEARCH PKT FORW. TABLE FOR DEST.
906
908
IS DESTIN. IN TABLE?
NO
YES
910
RETRIEVE LAYER 2 ADDRESS
912
MODIFY HEADER
914
TRANSMIT PACKET TO NEXT HOP DEST.
910
TRANSMIT PACKET TO ROUTER
912
ENTER PACKET DATA IN HISTORY TABLE
914
EXTRACT DEST. FROM HEADER
916
PACKET FROM SWITCH?
YES
NO
STORE LAYER 2 ADDRESS IN FORW. TABLE
918
REMOVE PKT DATA FROM HISTORY TABLE
920
TRANSMIT PACKET TO NEXT HOP DEST.
922
TRANSMIT PACKET TO NEXT HOP DEST.
924

Figure 9

DESTINATION IP ADDRESS	NEXT HOP IP ADDRESS
⋮	⋮
ANY OTHER ADDRESS	DEFAULT IP ADDRESS

Figure 10A

NEXT HOP IP ADDRESS	NEXT HOP PHYSICAL ADDRESS
⋮	⋮

Figure 10B

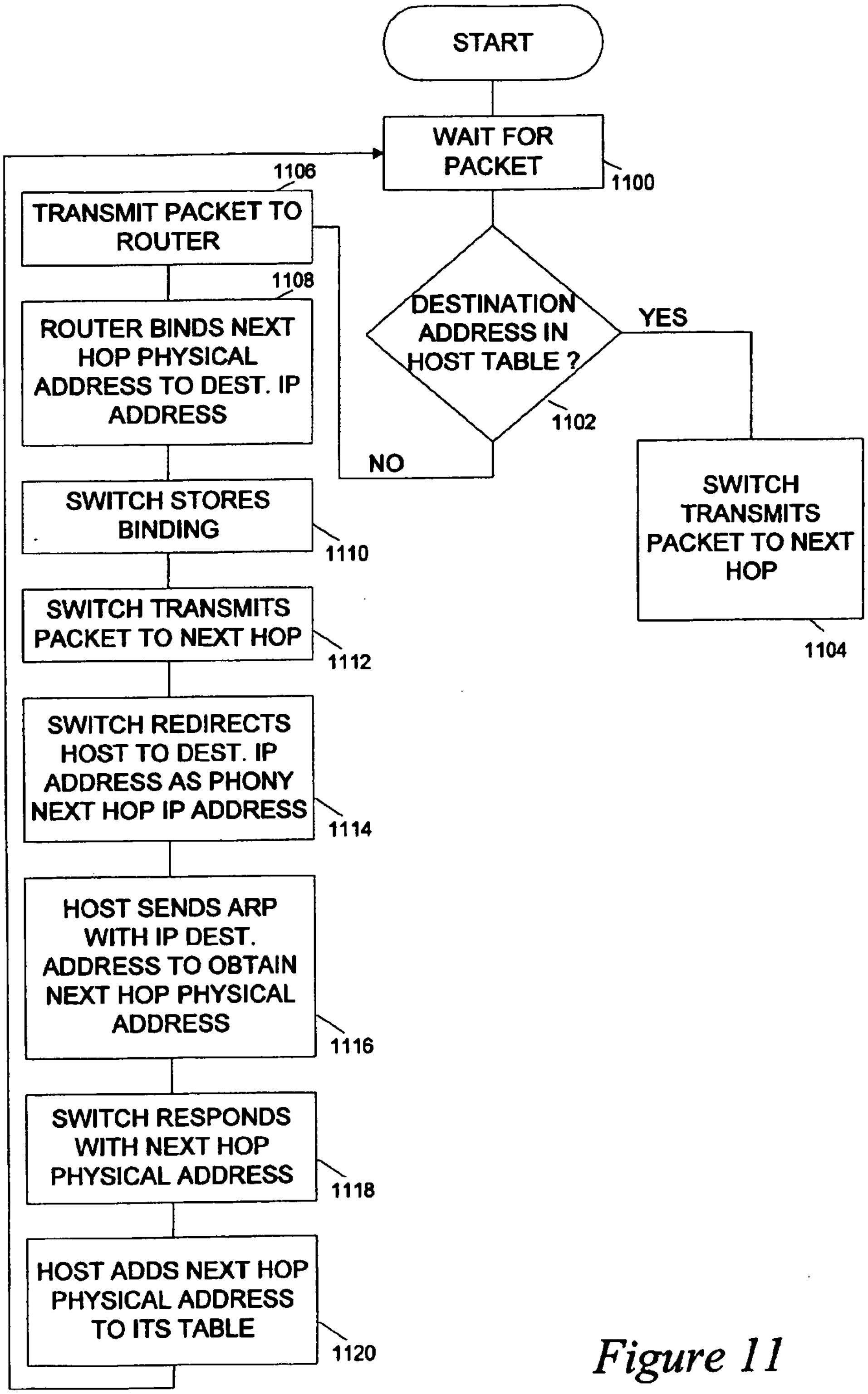

START
WAIT FOR PACKET
1100
DESTINATION ADDRESS IN HOST TABLE ?
1102
YES
NO
SWITCH TRANSMITS PACKET TO NEXT HOP
1104
TRANSMIT PACKET TO ROUTER
1106
ROUTER BINDS NEXT HOP PHYSICAL ADDRESS TO DEST. IP ADDRESS
1108
SWITCH STORES BINDING
1110
SWITCH TRANSMITS PACKET TO NEXT HOP
1112
SWITCH REDIRECTS HOST TO DEST. IP ADDRESS AS PHONY NEXT HOP IP ADDRESS
1114
HOST SENDS ARP WITH IP DEST. ADDRESS TO OBTAIN NEXT HOP PHYSICAL ADDRESS
1116
SWITCH RESPONDS WITH NEXT HOP PHYSICAL ADDRESS
1118
HOST ADDS NEXT HOP PHYSICAL ADDRESS TO ITS TABLE
1120

Figure 11

DESTINATION IP ADDRESS	NEXT HOP PHYSICAL ADDRESS

Figure 12

NEXT HOP IP ADDRESS	NEXT HOP PHYSICAL ADDRESS

Figure 14

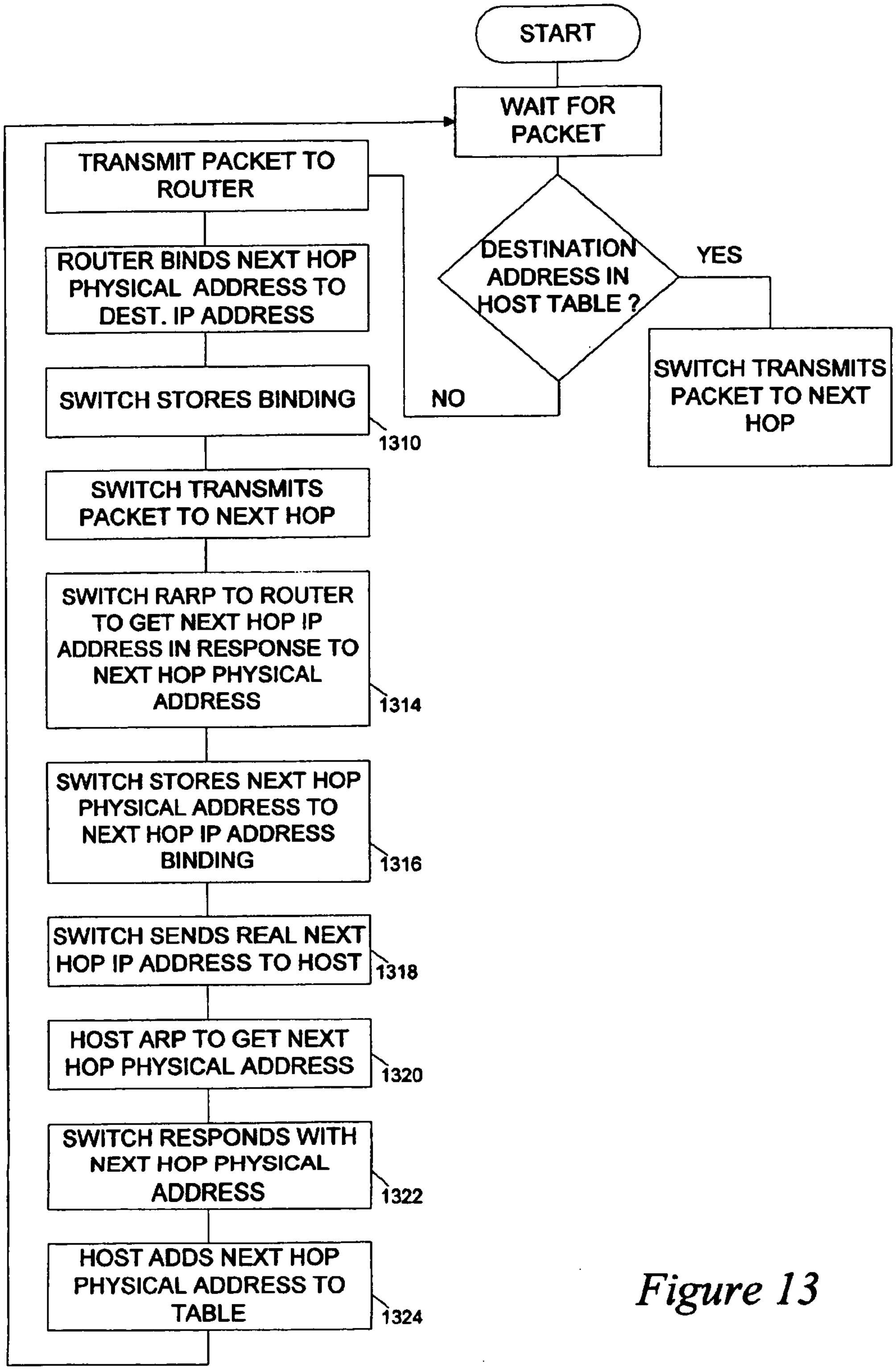

START
WAIT FOR PACKET
DESTINATION ADDRESS IN HOST TABLE ?
YES
SWITCH TRANSMITS PACKET TO NEXT HOP
NO
TRANSMIT PACKET TO ROUTER
ROUTER BINDS NEXT HOP PHYSICAL ADDRESS TO DEST. IP ADDRESS
SWITCH STORES BINDING
1310
SWITCH TRANSMITS PACKET TO NEXT HOP
SWITCH RARP TO ROUTER TO GET NEXT HOP IP ADDRESS IN RESPONSE TO NEXT HOP PHYSICAL ADDRESS
1314
SWITCH STORES NEXT HOP PHYSICAL ADDRESS TO NEXT HOP IP ADDRESS BINDING
1316
SWITCH SENDS REAL NEXT HOP IP ADDRESS TO HOST
1318
HOST ARP TO GET NEXT HOP PHYSICAL ADDRESS
1320
SWITCH RESPONDS WITH NEXT HOP PHYSICAL ADDRESS
1322
HOST ADDS NEXT HOP PHYSICAL ADDRESS TO TABLE
1324

Figure 13

METHOD AND APPARATUS FOR ACCELERATING OSI LAYER 3 ROUTERS

BACKGROUND

1. Field of the Invention

The present invention relates to the field of routing messages in a communications network, and in particular to increasing the throughput of a routing system.

2. Description of the Related Art

In recent years, there has been an exponential increase in the demand for bandwidth in communications networks. This increase is due to a variety of factors, including an increase in the number of users as the Internet moves towards becoming a mass communications medium, and an increase in bandwidth-intensive multimedia applications that integrate still images, video and speech with data. To keep up with this increasing demand, communications facilities must frequently be upgraded.

Data routers are critical components in data communications networks. Routers link physical networks along the path from a source node to a destination node. In a network employing packet switching, data packets are passed from router to router until they reach their final destination. This process is known as "next hop routing."

FIG. 1 illustrates a conventional communications network of the type that employs next hop routing. In this example, a router **100** couples three networks **102**. Each network **102** includes at least one host computer **104**. Software in the router **100** implements an Internet protocol (IP) routing algorithm that determines how to send an IP datagram across the networks. The Internet protocol operates at layer 3 of the well-known layered OSI standard. For a host **104** to transfer an IP datagram to another host **104**, the sender encapsulates the datagram in a physical frame ("packet"), wherein the destination IP address is mapped to a physical (layer 2) address. If the source and destination hosts lie within the same physical network, such as an Ethernet, then the source can send the datagram directly to the destination over the physical network.

FIG. 2 illustrates the format of an IP datagram **200** encapsulated in a physical frame **202**. The frame **202** includes a frame header **204** followed by frame data **206** and a cyclic redundancy check code **208**. The IP datagram **200**, which is encapsulated as frame data **206**, includes a datagram header **210** and datagram data **212**.

FIG. 3 illustrates the format of a physical frame header **204**, such as an Ethernet frame header. The frame header **204** includes a preamble **300**, a destination address **302**, a source address **304**, and a frame type field **306**. The preamble is employed for synchronization purposes. The destination address **302** contains the physical (layer 2) address of the destination node, which is unique to each hardware unit and is hard-coded into the hardware at the time of manufacture. The source address **304** is the physical address of the source node.

FIG. 4 illustrates the format of an IP datagram header **210**, showing only fields that are relevant to the present invention. The fields include a time to live field **400**, an IP source address **402**, and an IP destination address **404**. For further information regarding frame formatting and networking in general, please refer to D. Comer, *Internetworking With TCP/IP*," Volume 1, Prentice-Hall, Third Edition, 1995, which is incorporated by reference herein.

If the source and destination lie in different networks, then the router is employed to effect the transfer. Both hosts and routers maintain IP routing tables to determine where to send a datagram based upon the IP destination address. Typically a routing table contains pairs (N, R), where N is the IP address of a destination network or host, and R is the IP address of the next router along the path to network or host N. In this manner, a router need not know the complete path to a destination node, only the next hop. The next hop in a machine's routing table must lie in a physical network to which the machine connects directly. To assure selection of the proper next hop, routers exchange routing tables to keep track of changes in network configuration.

After executing a routing algorithm to obtain the next hop IP address in response to the ultimate destination address of a datagram, the router passes the datagram and the next hop address to network interface software in the router responsible for the physical network over which the datagram must be sent. The network interface software binds the next hop address to a physical address, forms a frame using that physical address, places the datagram in the data portion of the frame, and sends the resulting frame to a second (next hop) router over the physical network linking the next hop router to the first router. This process continues until the datagram reaches a final router that is connected directly to the same physical network as the destination. At that point, the final router will deliver the datagram using direct delivery.

After using the next hop address to find a physical address, the network interface software discards the next hop address. If the host is sending a sequence of datagrams to the same destination address, this process of determining the next hop IP address and then the next hop physical address is repeated, even though it appears very inefficient. As is well known in the art, the binding between the next hop IP address and the physical address is not saved because of the philosophy underlying the Internet protocol. The protocol builds an abstraction that hides the details of one network layer from another, thereby maintaining the distinction between the IP and physical address layers.

Because router functions, such as exchanging routing tables and executing routing algorithms require the flexibility of software to react to a changing environment, these functions are implemented in software rather than hardware. Thus, the capacity of routers is limited to the capacity of the microprocessors and the software that implement the necessary algorithms.

In order to increase data throughput, some routers have been designed in hardware to overcome the limitations of software. For example, Rekhter describes systems that contain local forwarding tables in switches, known as "forwarding information bases." Each table entry includes an ultimate destination address and a next hop destination address. The forwarding information base is constructed from information contained in a "routing information base" that in turn is constructed from routing information received from the network. For further information please refer to Y. Rekhter, "Inter-domain routing: EGP, BGP, and IDRP," in M. Steenstrup, *Routing in Communications Networks*, First Edition, 1995, pp. 99–133. Forwarding tables are also employed in U.S. Pat. No. 5,566,170 issued to Bakke et al. These and all other references referred to herein are incorporated by reference herein.

Further, other hardware-accelerated routers cache next hop information. Although these hardware routers increase throughput, replacing existing routers with these advanced routers is an expensive proposition. The expense is not just the cost of the router itself, but the administrative costs incurred in configuring the new router and the cost of associated network downtime. Accordingly, it is desired to find a more cost-effective way of upgrading existing routing systems to increase data capacity.

SUMMARY OF THE INVENTION

The present invention provides methods and apparatus for accelerating a router in a communications network. In one embodiment, a router accelerator includes a forwarding table for associatively storing a destination address and a next hop address. If a destination address of a packet matches a destination address in the forwarding table, then logic forwards the packet to a next hop. The destination address may be a network address, and the next hop address may be a physical address.

A router may be coupled to at least one network port through the router accelerator. The router binds the next hop address to the destination address of the packet if the packet destination address does not match a destination address in the forwarding table. The router does not process a packet if the packet destination matches in the forwarding table.

A router history table associatively stores at least one destination address and a corresponding address of the network port that received a packet having that destination address. Each forwarding table may be associated with a network port. The accelerator stores the next hop address in the forwarding table for a particular port if that port is associated in the history table with a destination address that is bound to the next hop address.

In another embodiment, the network includes at least one host and at least one router. The host has at least one routing table for associatively storing a second-level destination address, a second-level next hop address and a first-level next hop address. The router binds a first-level next hop address to a second-level destination address. A router accelerator includes redirect logic for storing the second-level destination address in a second-level next hop address entry in the at least one host routing table. This causes the host to request a first-level next hop address corresponding to the second-level destination address. An accelerator table associatively stores the bound second-level destination address and first-level next hop address. The accelerator table responds to the host's request with the bound first-level next hop address.

The first-level addresses may be physical addresses, and the second-level-addresses may be network addresses. The router may be coupled to at least one network port through the router accelerator.

The host associatively stores the bound second-level destination address and first-level next hop address in a host forwarding table. The host forwards a packet to a next hop if a second-level destination address of the packet matches a second-level destination address in the host forwarding table. A router coupled to the accelerator does not process the packet if the second-level destination address of the packet matches a second-level destination address in the host forwarding table.

In another embodiment employing host routing tables, a router accelerator includes request logic for requesting from the router a second-level next hop address in response to the first-level next hop address. Redirect logic stores the second-level next hop address in a second-level next hop address entry in the at least one host routing table. This causes the host to request a first-level next hop address corresponding to the second-level next hop address. An accelerator tables associatively stores the corresponding second-level next hop address and first-level next hop address. The accelerator table responds to the host's request with the bound first-level next hop address.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a conventional communications network of the type that employs next hop routing.

FIG. 2 illustrates the format of an IP data gram in capsulated in a physical frame.

FIG. 3 illustrates the format of a physical frame header.

FIG. 4 illustrates the format of a IP datagram header.

FIG. 5 is a communications network incorporating a router accelerator switch of the present invention.

FIG. 6 illustrates a router accelerator switch according to the first embodiment to the present invention.

FIG. 7 illustrates the data structure for a packet forwarding table of the present invention.

FIG. 8 illustrates the data structure of a router history table of the present invention.

FIG. 9 is a flow chart illustrating the operation of the router accelerator switch of the present invention.

FIG. 10A illustrates a first host routing table employed by the present invention.

FIG. 10B illustrates a second routing table employed by the present invention.

FIG. 11 is a flow chart illustrating the operation of a second embodiment of the present invention.

FIG. 12 illustrates an accelerator table employed by the host-syntric embodiments of the present invention.

FIG. 13 is a flow chart illustrating the operation of a third embodiment of the present invention.

FIG. 14 illustrates a RARP table of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The present invention provides methods and apparatus for accelerating OSI layer 3 routing. In the following description, numerous details are set forth in order to enable a thorough understanding of the present invention. However, it will be understood by those of ordinary skill in the art that these specific details are not required in order to practice the invention. Further, well-known elements, devices, process steps and the like are not set forth in detail in order to avoid obscuring the present invention.

The present invention provides a means whereby existing routing systems may be upgraded to increase throughput at low cost. FIG. 5 illustrates a communications network incorporating a router accelerator switch 500 of the present invention. The networks 502, the switch 500 and the router 504 may form a wide-area network that connects a number of geographically-dispersed networks together. Alternatively, the networks 502 may be local-area networks at a given site, perhaps in different buildings or of different types, such as a fiber-optic token ring network or an Ethernet. In the description that follows, it will be assumed for simplicity that the networks are Ethernet local area networks. Further, although three networks 502 are illustrated, many more networks would be connected in most actual installations.

The router accelerator switch 500 of the present invention is independent of the design of the router 504, the routing algorithms and the network topology. The accelerator 500 may be installed in a conventional routing system by simply unplugging the router 504 from the network connections, connecting the accelerator to those connections, and plugging the router 504 into the accelerator switch 500. Through the operation described below, the accelerator switch 500 is transparent to both the networks 502 and the router 504, except that the router 504 will appear to have a much larger capacity for switching packets.

The distinction between a router and a switch is somewhat arbitrary, but generally lies in the level of sophistication of the mechanisms used to determine the next hop destination, and the extent to which the system communicates with other parts of the network to exchange information from routing tables. For purposes of this application, a router performs the function of calculating a next hop destination by communicating with other parts of the network to monitor the network configuration. Conversely, for purposes of the application, a conventional switch is assumed not to perform any calculations to determine the route of a packet, but simply forwards the packet according to next hop information provided by the router.

According to the present invention, the switch of the invention is connected between the network ports and the routing system in order to intercept packets transmitted to and from the router. The switch acts as a "shadow router" to observe and mimic some of the router's functions. By doing so, the switch need not receive instructions from the router in order to route every packet.

In one embodiment, the switch records the next hop physical address computed by the router for a packet. The switch then usurps the next hop routing function of the router for subsequent packets directed to the same destination.

FIG. 6 illustrates a router accelerator switch 500 according to a first embodiment of the present invention. The switch includes three ports 600 connected to three corresponding networks (not shown), and three ports 602 connected to corresponding ports of a router (not shown). The ports correspond one-to-one with each other to maintain the transparency of the switch. Each port includes packet processing logic (PPL) (604 on network side, 606 on router side) for handling layer 2 and layer 3 communications with, for example, an Ethernet network. The PPL circuitry may be implemented using standard Ethernet interface chips.

Each port 600 on the network side is coupled to a first content addressable memory 608 (CAM) that implements a packet forwarding table (PFT) of the invention. On the router side, the switch includes a table 610, implemented with a CAM, which acts as a router history table of the invention. Shared memory packet storage 612 acts as a queue for storing packets, with one queue per router/network port. The packet storage is shared by both the router ports and the network ports. A microprocessor 614 executes programs stored in flash memory 616 and uses DRAM memory 618 for temporary storage. A memory controller 620 controls transfer of the packets among the router ports 602, the network ports 600 and the shared memory 612. The memory controller 620 keeps track of the location of packets in memory. It is also responsible for notifying the microprocessor of an overflow. Control signals are transferred throughout the switch over a control bus 622. Data is transferred between the shared memory 612 and the network ports 600 over a first data bus 624, whereas data is transferred between the router ports 602 and shared memory 612 over a second data bus 626.

FIG. 7 illustrates the data structure for the packet forwarding table 608. The packet forwarding table 608 associatively stores the IP address of the ultimate destination of a packet, the layer 2 (physical) address of the next hop, an indicator of the network-side port linked to the next hop (destination port), and a time to exist (aging) field indicating how long the address entry is to remain in the table. Because the switch must be transparent to the router and the network, the destination port is the same port at which the packet was received from the router side.

FIG. 8 illustrates the data structure of the router history table 610. The router history table 610 associatively stores the IP destination address along with the port at which the router received a packet having the associated IP destination address.

Referring to the flow chart of FIG. 9, the router accelerator switch of the present invention operates as follows. The switch waits for a packet to arrive either at a network-side port or a router-side port (step 900). When a packet is received over one of the network ports (step 902) as being directed to the router as a next hop, it is intercepted by the router accelerator switch at the switch's associated network port input. The PPL 604 then extracts the IP destination address from the packet frame (step 904) and forwards it to the port's packet-forwarding table for comparison with the IP destination addresses stored in the CAM of the PFT (steps 906, 908). The packet processing logic 604 also instructs the memory controller to store the packet in the shared memory packet storage.

If the IP destination address does not match an address in the packet forwarding table, then the packet processing logic instructs the memory controller over the control bus to transmit the packet from shared memory to the corresponding router port over the router-side data bus, so that the packet may be processed by the router (step 910).

Upon receiving the packet, the router performs its usual function of computing the next hop layer 2 (physical) address. After doing so, the router transmits the packet back to the switch over the appropriate router-side port leading to the next hop destination (steps 900, 902). The packet returned to the switch by the router is encapsulated in a physical frame having the next hop layer 2 address in the frame header. The router-side packet processing logic 606 of the switch at the port receiving the returning packet instructs the memory controller to store this packet in shared memory. Further, under instructions from the router-side packet processing logic 606, the memory controller also causes the IP destination address and input port identifier to be stored in the router history table (step 912).

The router-side PPL 606 then extracts the IP destination address from the packet header (step 914). The PPL then causes the destination address to be compared to the addresses in the router history table (step 916). If there is a hit, then this indicates that the packet was previously sent to the router by the router accelerator switch.

In response to a hit, the network-side packet processing logic instructs the memory controller to extract the IP destination address and the layer 2 next hop address from the packet stored in shared memory. The memory controller then stores this data in the packet forwarding table (or tables) for the ports indicated by the router history table as corresponding to the associated IP destination address (or addresses) (step 918). After the next hop information is stored in the packet forwarding table, the router history table deletes the entry for the corresponding destination address (step 920).

The router-side packet processing logic 606 forwards the returned packet to the next hop destination through the network port corresponding to the router port serving as output, which is not necessarily the network port from which the packet was received (step 922).

At this point, the packet forwarding table now stores the address binding for a first packet in a stream of packets directed towards a particular destination, where the packet has been processed by the router. If the next received packet is from the network, then the network-side packet processing logic 604 for the input port causes the memory controller 620 to store the packet in shared memory 612 and to extract the destination address from the IP datagram header (steps 900, 902, 904). The packet processing logic then causes the IP destination address to be input to the associated packet forwarding table (step 906). If the table indicates a hit, then the table transfers the associated physical (layer 2) next hop address to the packet processing logic (steps 908, 910). The packet processing logic causes the memory controller to forward the IP datagram to the packet processing logic 604, which encapsulates the datagram in a frame including the layer 2 next hop address (step 912). The packet processing logic 604 then forwards the frame to the next hop destination on the appropriate output port (step 914).

In some instances, a packet is originated at the router (steps 902, 904). Such packets typically convey status information, such as data to update routing tables. The router encapsulates these packets in a physical frame specifying the next hop destination. Router-generated packets are not stored in the router history table. The router-side packet processing logic that received the router-generated packet stores the frame in a shared memory queue. The memory is logically subdivided into a queue for each output port. When it is found by the router-side PPL that the packet originated from the router, then the PPL forwards the packet to shared memory. When the packet reaches the top of the queue, it is transmitted from the corresponding output port (step 924).

Packets addressed directly to the router's IP address from the network are sent by the receiving port's network-side PPL to the router, but not entered in the router history table because it is not expected that the router will return them.

The first embodiment described above relies entirely on tables stored in the switch. The following second and third embodiments may be deemed "host-centric" because they store relevant routing information in pre-existing routing tables in the host.

The host-centric router accelerator relies upon host routing tables that can perform the address resolution protocol (ARP) function. To perform ARP, a host sends out a broadcast message to all nodes within its broadcast domain, i.e., local area network. The ARP message contains an IP destination address and a request for the node having that address to transmit back its physical address. In this manner, the host binds an IP address to a physical address.

FIGS. 10A and 10B illustrate relevant entries in two routing tables found in a conventional host. The first table associatively stores a destination IP (second-level) address along with a corresponding next hop IP (second-level) address. The second table, known as the host ARP cache or forwarding table, associatively stores a next hop IP (second-level) address along with the corresponding next hop physical (first-level) address. These two tables together effectively provide a mapping of destination IP address to next hop physical address. The reason two tables are employed is that the address boundary between the layers is maintained by using this format.

The operation of one host-centric embodiment is illustrated in FIG. 11. The structure of the router accelerator is essentially the same as that of FIG. 6, except the accelerator does not include the packet forwarding tables, and the router-side table in both host-centric embodiments performs a different function than the router history table, as explained below.

Generally, packets originate at a host. The host determines whether the destination IP address specified by the packet is found in its own host tables (step 1102). If so, then the host uses its ARP cache to send the packet directly to the next hop physical address (step 1104).

If no match, the host forwards the packet to a default router through the intervening router acceleration switch of the invention (step 1106). At this point, the switch just passes the packet to the router.

As before, the router binds the next hop physical address to the destination IP address (step 1108). The router then sends the IP datagram encapsulated in a physical frame with the next hop physical address into the switch router-side port leading to the next hop destination. The packet processing logic 606 at the port 602 then stores the frame in shared memory 612 over the router-side data bus 626. The router-side packet processing logic 606 also instructs the memory controller 620 to output the frame from the shared memory 612 onto the network-side data bus 624, and instructs the packet processing logic 604 of the network port corresponding to the router port to forward it to the next hop (step 1112).

Before the switch transmits the packet to the next hop, the microprocessor 614 of the accelerator switch stores the destination IP/next hop physical address binding in the router-side table 610 (the "ARP table" in this embodiment) (step 1110). The table for storing this binding is shown in FIG. 12, where the destination IP and next hop physical addresses are associatively stored.

The switch then takes steps to "trick" the host forwarding tables into storing this binding. The switch microprocessor 614 employs the Internet control message protocol (ICMP) to perform this function. The ICMP message is encapsulated in an IP datagram, which is itself encapsulated in a frame for transmission. One type of ICMP message is a redirect message. In conventional systems, routers employ redirect messages to update host routing tables. A host boots up knowing the address of only one router on the local network. This default router returns an ICMP redirect message whenever a host sends a datagram for which there is an initial router along a more optimal path. The redirect message replaces the default router in the table with the address of this more optimal router.

According to the present invention, the microprocessor in the switch sends a redirect message to the host to cause the first host table to associate the destination IP address of the packet with itself so as to act as the next hop IP address in the table (step 1114). Based upon this table entry, the host sends an ARP message to obtain the corresponding physical address (step 1116). In a conventional system, the ARP message would be used to obtain a physical address corresponding to an IP address in the table. Here, because the destination IP address has been substituted for the next hop IP address, a conventional system would attempt to return a physical destination address. However, in a conventional system this would only work if the destination node were within the same local network (broadcast domain) as the host. In contrast, the present invention employs the ARP table of the switch to intercept and respond to the host's ARP request. Recall that the ARP table has already stored the address binding computed by the router. In this manner, the switch acts as an ARP server to return the next hop physical address in response to the IP destination address sent by the host (step **1118**). The host then adds this next hop physical address to its ARP cache, the second host table (step **1120**).

The switch then waits for the next packet (step **1100**). The microprocessor **614** in the switch determines whether the destination address specified by this packet matches a destination address in the ARP cache (step **1102**). If so, the host responds with the next hop physical address. The microprocessor, coupled to the host network, then forwards the packet to the next hop destination at layer **2** (step **1104**). Subsequent packets directed to the same destination IP address are handled in a similar manner. Through this technique, the host takes much of the burden off of the default router for sending packets to the first hop.

Now that the host knows the next hop IP address and the physical address of the next hop, it will send subsequent packets having that destination IP address directly to the physical address of the next hop. This process can be handled by the conventional layer **2** switching of the switch, which is found in the router-side packet processing logic **606**, microprocessor **614** and memory controller **620**.

Unlike the previous embodiment, this approach uses existing memory in the host to accomplish functions that would otherwise require memory in the switch. The switch need not store information for every host. Instead, the information is distributed among the hosts. Moreover, the switch only needs the address information one time for the ARP table, instead of maintaining the information for every packet it sends (as in the packet forwarding table of the previous embodiment). Finally, the switch does not need an high-speed memory like a CAM. Instead, it employs a software table for its ARP response function.

The operation of a second host-centric embodiment is illustrated in FIG. **13**. As can be seen from a comparison with FIG. **11**, many of the initial steps are the same. As before, the switch stores the destination IP address/next hop physical address binding in the ARP table in DRAM (step **1310**). However, in this embodiment, the switch also employs the reverse address resolution protocol (RARP).

The microprocessor **614** in the switch issues to the router a RARP request containing the next hop physical address in order to obtain the next hop IP address from the router (step **1314**). According to RARP, a machine on the network sends out a broadcast message containing a physical address to nodes on its local network in order to obtain a corresponding IP address. Conventional routers can act as RARP servers. The microprocessor in the switch stores the RARP next hop physical address/next hop IP address binding in the table illustrated in FIG. **14**, which is also stored in DRAM (step **1316**).

Unlike the previous embodiment, the switch sends the real next hop IP address to the first host table (step **1318**). The host issues an ARP request with this next hop IP address in order to obtain the next hop physical address (step **1320**). The switch acts as an ARP server to return the next hop physical address (step **1322**). The host adds the next hop physical address to its ARP cache (step **1324**). Subsequent packets directed to the same destination IP address then match in the host tables. Again, by filling the host tables with the next hop physical address information, the host can facilitate layer **2** switching, therefore bypassing the router.

The RARP approach does not require the counterfeit redirect message of the previous embodiment. The redirect message of the previous embodiment may be rejected by Internet security systems that are sensitive to such ARP spoofing.

Based on the foregoing, it can be seen that the router accelerator of the present invention need not receive and maintain information about the status of the network or execute routing algorithms. Thus, packet forwarding for the vast majority of packets in a source-destination stream can be performed by the high-speed router accelerator, which is implemented entirely in hardware. Further, performance of the router accelerator is not hampered by the need to maintain and update software for implementing routing algorithms as the algorithms change.

Although the invention has been described in conjunction with particular embodiments, it will be appreciated that various modifications and alterations may be made by those skilled in the art without departing from the spirit and scope of the invention. The invention is not to be limited by the foregoing illustrative details, but rather is to be defined by the amended claims.

What is claimed is:

1. A router accelerator switch comprising:

a forwarding table for associatively storing a destination address and a next hop address;

logic for forwarding a packet to a next hop if a destination address of the packet matches a destination address in the forwarding table, wherein an associated router does not process the packet if the packet destination address matches a destination address in the forwarding table, and logic for communicating the packet to the router if the packet destination address does not match a destination address in the forwarding table, wherein the router computes the next hop address corresponding to the destination address of the packet if the packet destination address does not match a destination address in the forwarding table.

2. The router accelerator switch of claim **1**, wherein the destination address is a network address and the next hop address is a physical address.

3. The router accelerator switch of claim **1**, wherein the router is coupled to at least one network port through the router accelerator switch.

4. The router accelerator switch of claim **3**, wherein the router binds the next hop address to the destination address of the packet if the packet destination address does not match a destination address in the forwarding table.

5. The router accelerator switch of claim **1**, further comprising a router history table for associatively storing at least one destination address and a corresponding address of a network port that received a packet having that destination address.

6. The router accelerator switch of claim **5**, wherein each network port has an associated forwarding table, and the accelerator stores the next hop address in the forwarding table for a port if the port is associated in the history table with a destination address that is bound to the next hop address.

7. In a network having at least one host and at least one router, the host having at least one routing table for associatively storing a second-level destination address, a second-level next hop address and a first-level next hop address, the router for binding a first-level next hop address to a second-level destination address, a router accelerator switch comprising:

redirect logic for storing the second-level destination address in a second-level next hop address entry in the at least one host routing table to thereby cause the host to request a first-level next hop address corresponding to the second-level destination address;

an accelerator table for associatively storing the bound second-level destination address and first-level next hop address, wherein the accelerator table responds to the host's request with the bound first-level next hop address.

8. The router accelerator switch of claim 7, wherein first-level addresses are physical addresses and second-level addresses are network addresses.

9. The router accelerator switch of claim 7, wherein the router is coupled to at least one network port through the router accelerator.

10. The router accelerator switch of claim 7, wherein the host associatively stores the bound second-level destination address and first-level next hop address in a host forwarding table.

11. The router accelerator switch of claim 10, wherein the host forwards a packet to a next hop if a second-level destination address of the packet matches a second-level destination address in the host forwarding table.

12. The router accelerator switch of claim 10, wherein a router coupled to the accelerator does not process the packet if the second-level destination address of the packet matches a second-level destination address in the host forwarding table.

13. In a network having at least one host and at least one router, the host having at least one routing table for associatively storing a second-level destination address, a second-level next hop address and a first-level next hop address, the router for binding a first-level next hop address to a second-level destination address, a router accelerator switch comprising:

request logic for requesting from the router a second-level next hop address in response to the first-level next hop address;

redirect logic for storing the second-level next hop address in a second-level next hop address entry in the at least one host routing table to thereby cause the host to request a first-level next hop address corresponding to the second-level next hop address;

an accelerator table for associatively storing the corresponding second-level next hop address and first-level next hop address, wherein the accelerator table responds to the host's request with the bound first-level next hop address.

14. The router accelerator switch of claim 13, wherein first-level addresses are physical addresses and second-level addresses are network addresses.

15. The router accelerator switch of claim 13, wherein the router is coupled to at least one network port through the router accelerator.

16. The router accelerator switch of claim 13, wherein the host associatively stores the bound second-level destination address and first-level next hop address in a host forwarding table.

17. The router accelerator switch of claim 16, wherein the host forwards a packet to a next hop if a second-level destination address of the packet matches a second-level destination address in the host forwarding table.

18. The router accelerator switch of claim 13, wherein a router coupled to the accelerator does not process the packet if the second-level destination address of the packet matches a second-level destination address in the host forwarding table.

19. A method for accelerating a router comprising the steps of:

associatively storing a destination address and a next hop address in a forwarding table of a router accelerator switch; and

forwarding a packet to a next hop if a destination address of the packet matches a destination address in the forwarding table, wherein an associated router does not process the packet if the packet destination address matches a destination address in the forwarding table, and communicating the packet to the router if the packet destination address does not match a destination address in the forwarding table, wherein the router computes the next hop address corresponding to the destination address of the packet if the packet destination address does not match a destination address in the forwarding table.

20. The method of claim 19, wherein the destination address is a network address and the next hop address is a physical address.

21. The method of claim 19, wherein the router is coupled to at least one network port through the router accelerator.

22. The method of claim 21, further comprising the step of the router binding the next hop address to the destination address of the packet if the packet destination address does not match a destination address in the forwarding table.

23. The method of claim 19, further comprising the step of associatively storing in a router history table at least one destination address and a corresponding address of a network port that received a packet having that destination address.

24. The method of claim 23, wherein each network port has an associated forwarding table, the method further comprising the step of storing the next hop address in the forwarding table for a port if the port is associated in the history table with a destination address that is bound to the next hop address.

25. In a network having at least one host and at least one router, the host having at least one routing table for associatively storing a second-level destination address, a second-level next hop address and a first-level next hop address, the router for binding a first-level next hop address to a second-level destination address, a method for accelerating the router comprising the steps of:

storing the second-level destination address in a second-level next hop address entry in the at least one host routing table to thereby cause the host to request a first-level next hop address corresponding to the second-level destination address; and

associatively storing in an accelerator table the bound second-level destination address and first-level next hop address, wherein the accelerator table responds to the host's request with the bound first-level next hop address.

26. The method of claim 25, wherein first-level addresses are physical addresses and second-level addresses are network addresses.

27. The method of claim 25, wherein the router is coupled to at least one network port through the router accelerator.

28. The method of claim 25, wherein the host associatively stores the bound second-level destination address and first-level next hop address in a host forwarding table.

29. The method of claim 28, wherein the host forwards a packet to a next hop if a second-level destination address of the packet matches a second-level destination address in the host forwarding table.

30. The method of claim 28, wherein a router coupled to the accelerator does not process the packet if the second-level destination address of the packet matches a second-level destination address in the host forwarding table.

31. In a network having at least one host and at least one router, the host having at least one routing table for asso-

ciatively storing a second-level destination address, a second-level next hop address and a first-level next hop address, the router for binding a first-level next hop address to a second-level destination address, a method for accelerating the router comprising the steps of:

requesting from the router a second-level next hop address in response to the first-level next hop address;

storing the second-level next hop address in a second-level next hop address entry in the at least one host routing table to thereby cause the host to request a first-level next hop address corresponding to the second-level next hop address;

associatively storing in an accelerator table the corresponding second-level next hop address and first-level next hop address, wherein the accelerator table responds to the host's request with the bound first-level next hop address.

32. The method of claim **31**, wherein first-level addresses are physical addresses and second-level addresses are network addresses.

33. The method of claim **31**, wherein the router is coupled to at least one network port through the router accelerator.

34. The method of claim **31**, wherein the host associatively stores the bound second-level destination address and first-level next hop address in a host forwarding table.

35. The method of claim **34**, wherein the host forwards a packet to a next hop if a second-level destination address of the packet matches a second-level destination address in the host forwarding table.

36. The method of claim **31**, wherein the router does not process a packet if the second-level destination address of the packet matches a second-level destination address in the host forwarding table.

* * * * *

US005473607A

United States Patent [19]

Hausman et al.

[11] **Patent Number: 5,473,607**

[45] **Date of Patent: Dec. 5, 1995**

[54] **PACKET FILTERING FOR DATA NETWORKS**

[75] Inventors: **Richard J. Hausman**, Soquel; **Lazar Birenbaum**, Saratoga, both of Calif.

[73] Assignee: **Grand Junction Networks, Inc.**, Union City, Calif.

[21] Appl. No.: **103,659**

[22] Filed: **Aug. 9, 1993**

[51] **Int. Cl.⁶** **H04L 12/28**

[52] **U.S. Cl.** **370/85.13**; 370/94.1; 370/92

[58] **Field of Search** 370/60, 60.1, 85.13, 370/85.14, 94.1, 94.2, 94.3, 92, 85.3; 395/159, 118, 400

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,399,531	8/1983	Grande et al.	370/60
4,627,052	12/1986	Homre et al.	370/85.13
4,679,193	7/1987	Jensen et al.	370/94.1
4,891,803	1/1990	Juang et al.	370/60
4,933,937	6/1990	Konishi	370/85.13
5,032,987	7/1991	Broder et al.	364/200
5,210,748	5/1993	Onishi et al.	370/60
5,218,638	6/1993	Matsumoto et al.	380/23
5,247,620	9/1993	Fukuzawa et al.	370/85.13
5,274,631	12/1993	Bhardwaj	370/60

OTHER PUBLICATIONS

Kalpana, Etnerswithc Product Overview, Mar. 1990, pp. 1–20.

Artel, Galactica Stenbridg C/802.3 Application Note, Nov. 1991, pp. 1–26.

Synemetics, Lanplex 5000: Intra–Network Banowidth, 1992, pp. 1–12.

Synemetics, Lanplex 5000 Intelligent Switching Hubs, 1993, pp. 1–9.

Synemetics, Lanplex 5000 Family, 1992. pp. 1–6.

Synemetics, Etheract Express Module, 1991, pp. 1–4.

Alantec, Powerhub Arthitectures, Dec. 1992, pp. 1–6.

Bradner, Scott O., "Ethernet Bridges and Routes", Feb. 1992, pp. 1–10, Data Communications.

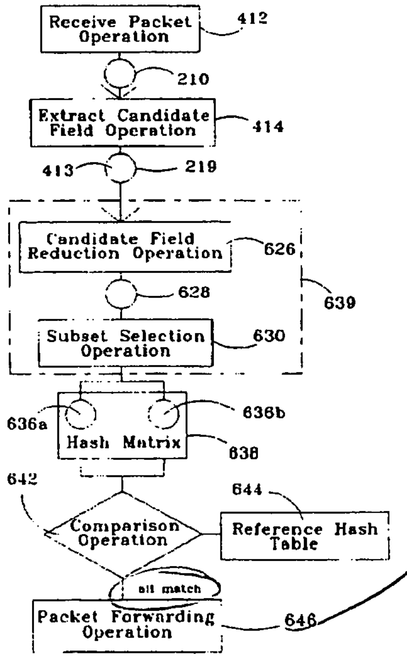
Kwok, Conrad K. and Biswanath Mjkherjec, "Cut–Through Bridging for CSMA/CE Local Area Network", Jul. 1990, pp. 938–942, I EEE Transactions on Communications, vol. 38, No. 7.

Primary Examiner—Douglas W. Olms
Assistant Examiner—Chau T. Nguyen
Attorney, Agent, or Firm—Michael J. Hughes

[57] **ABSTRACT**

An improved partial packet filter (**10**) for filtering data packets (**210**) in a computer network (**12**) wherein a candidate field (**413**) of the data packet (**210**) is hashed to a plurality of bit-wise subsets (**636**) each being an independent representation of the candidate field (**413**). Each of the bit-wise subsets (**636**) is compared to a reference hash table (**644**) which has been prepared in a preliminary operation series (**514**). The preliminary operation series (**512**) configures a plurality of target fields (**714**) to set selected memory locations (**312**) in the reference hash table (**644**).

17 Claims, 4 Drawing Sheets



extracting values from packet
hashing
comparing to table
match
forward

Fig. 1

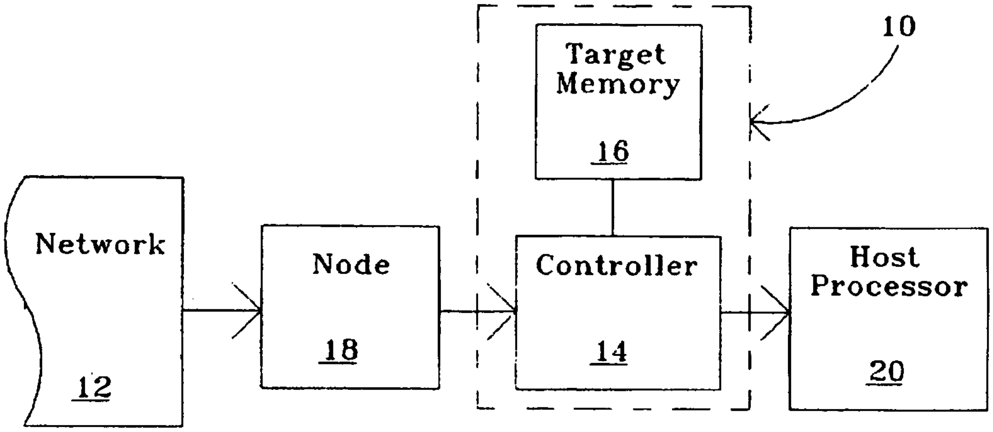

Target Memory
16
10
Network
12
Node
18
Controller
14
Host Processor
20

Fig. 2

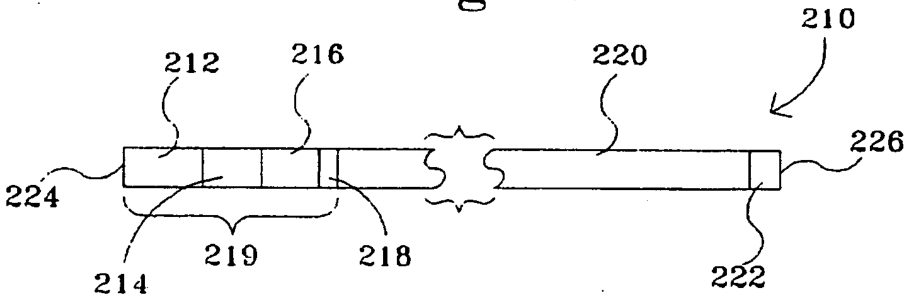

210
212
216
220
226
224
219
218
214
222

Fig. 3

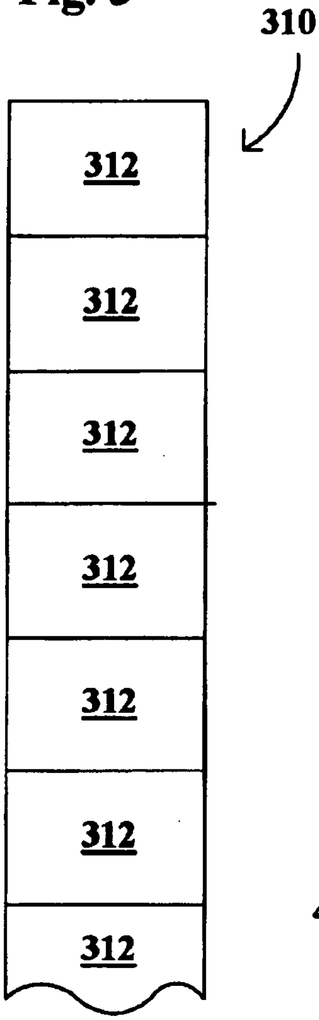


Fig. 4 (prior art)

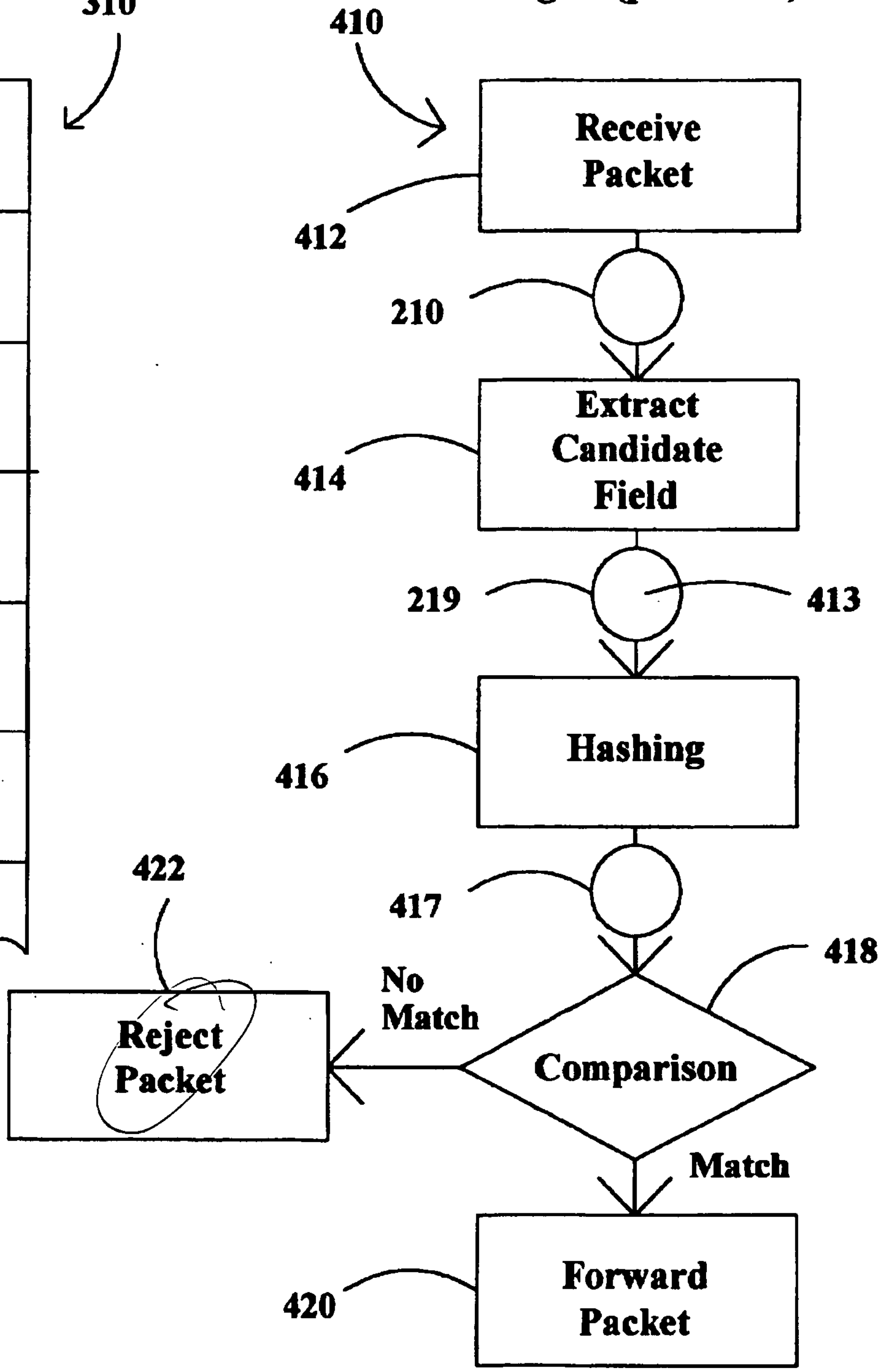


Fig. 5

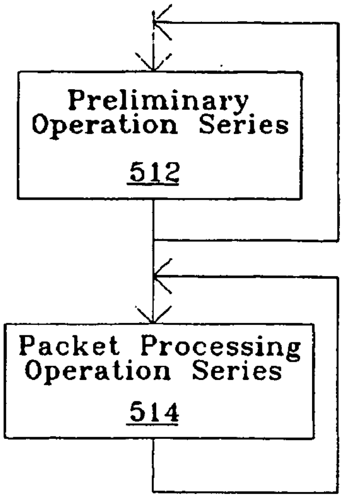

Preliminary Operation Series
512
Packet Processing Operation Series
514

Fig. 6

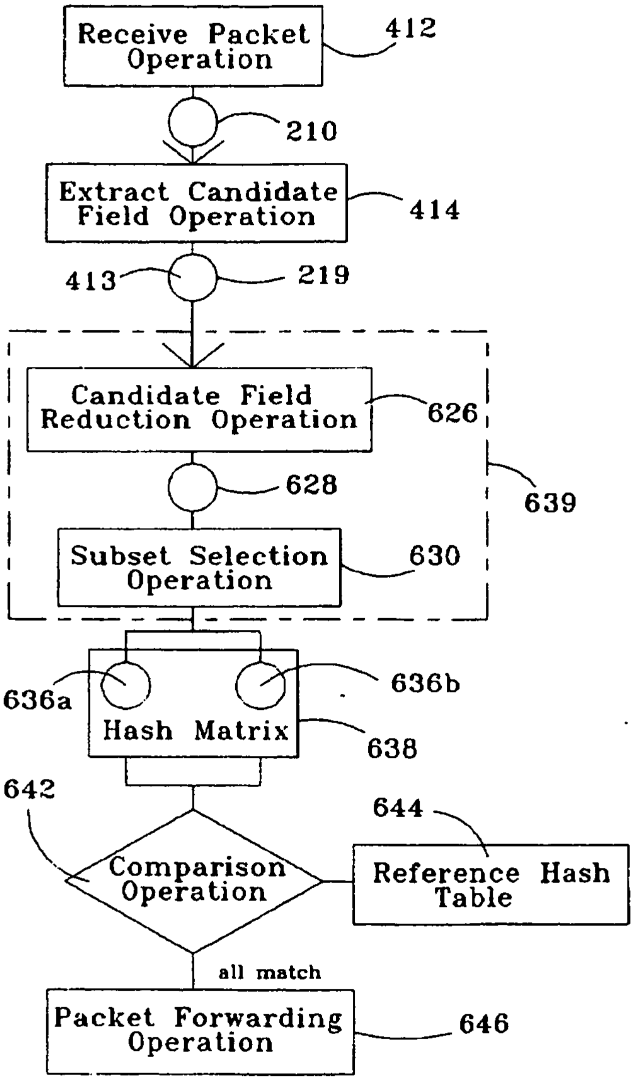
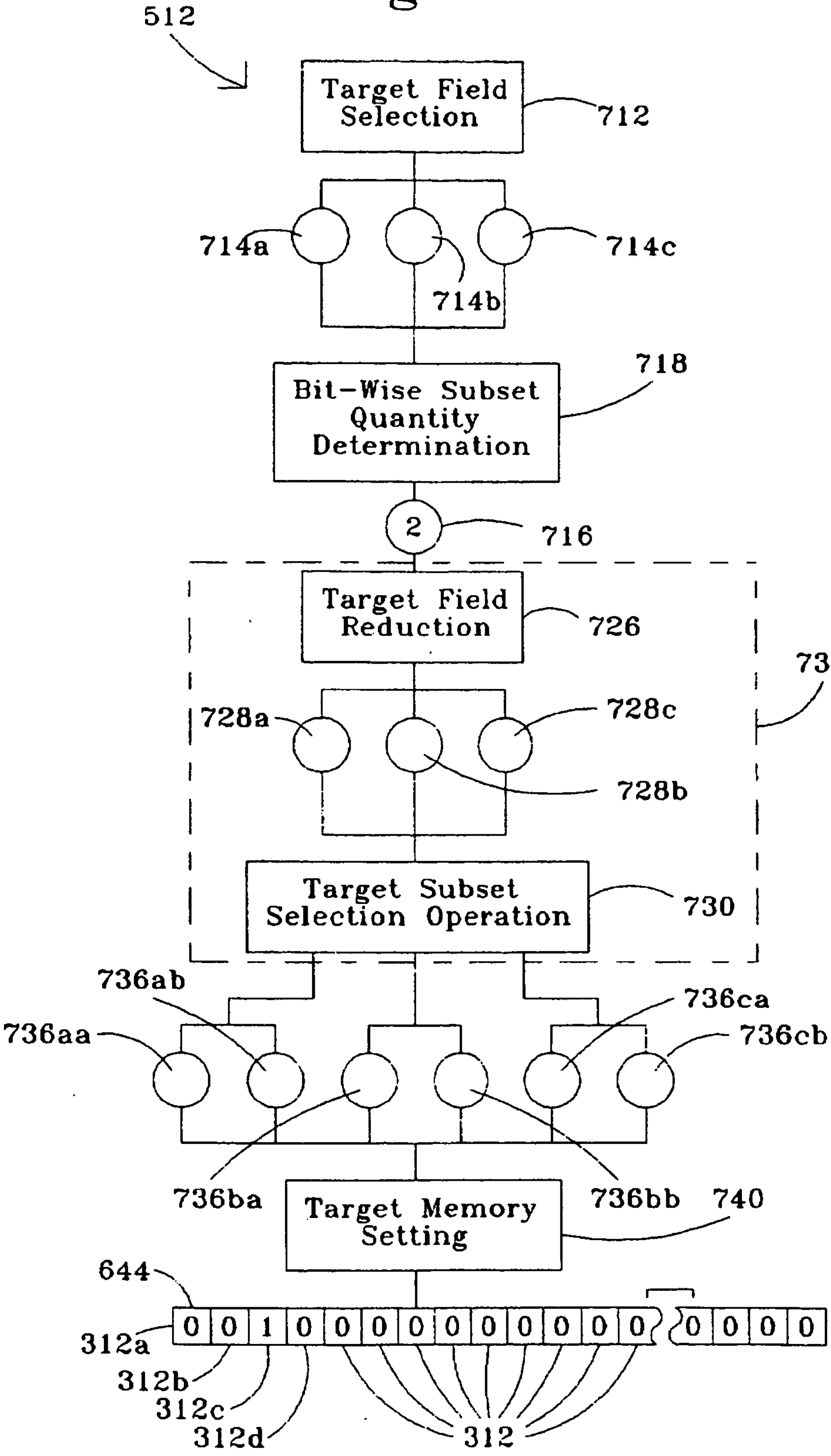

Receive Packet Operation
412
210
Extract Candidate Field Operation
414
413
219
Candidate Field Reduction Operation
626
639
628
Subset Selection Operation
630
636a
636b
Hash Matrix
638
642
644
Comparison Operation
Reference Hash Table
all match
Packet Forwarding Operation
646

Fig. 7


512
Target Field Selection
712
714a
714b
714c
718
Bit-Wise Subset Quantity Determination
2
716
Target Field Reduction
726
739
728a
728c
728b
Target Subset Selection Operation
730
736ab
736ca
736aa
736cb
736ba
736bb
740
Target Memory Setting
644
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
312a
312b
312c
312d
312

PACKET FILTERING FOR DATA NETWORKS

TECHNICAL FIELD

The present invention relates generally to the field of computer science and more particularly to data networking and component devices attached to data networks.

BACKGROUND ART

Computer networks are becoming increasingly common in industry, education and the public sector. The media over which data are carried generally carry data in units referred to as "packets" which are destined for many different sources. Addressing and packet typing are included in most standardized and proprietary packet based networking protocols which make use of destination address fields at the beginning of and/or within each data packet for the purpose of distinguishing proper recipient(s) of the data of the packets. As a packet is received at intermediate and end components in a system, rapid determination of the proper recipient (s) for the data must be made in order to efficiently accept, forward, or discard the data packet. Such determinations are made based upon the above discussed address, packet type and/or other fields within the relevant packets. These determinations can be made by network controller hardware alone, by a combination of hardware and software, or by software alone. In broadcast type networks, every node is responsible for examining every packet and accepting those "of interest", while rejecting all others. This is called "packet filtering". Accuracy, speed and economy of the filtering mechanism are all of importance.

When the above discussed determinations are made through a combination of hardware and software, the hardware is said to have accomplished a "partial filtering" of the incoming packet stream. It should be noted that one type of packet filtering is accomplished on the basis of packet error characteristics such as collision fragments known as "runts", frame check sequence errors, and the like. The type of filtering relevant to the present discussion is based upon packet filtering in which filtering criteria can be expressed as simple Boolean functions of data fields within the packet as opposed to filtering based upon detection of errors or improperly formed packets.

In the simplest case, each node of a computer network must capture those packets whose destination address field matches the node's unique address. However there frequently occur situations in which additional packets are also of interest. One example occurs when the node belongs to a predefined set of nodes all of which simultaneously receive certain specific "groupcast" packets which are addressed to that group. Groupcast packets are usually identified by some variation of the address field of the packet. Groupcast address types generally fall into one of two forms. "Broadcast" addresses are intended for all nodes and "multicast" addresses are targeted for specific applications to which subsets of nodes are registered. Another case of such field-based packet filtering occurs when certain network management nodes are adapted to focus on specific protocols, inter-node transactions, or the like, to the exclusion of all other traffic.

Attachment of a networked device to the network is realized through a "controller" which operates independently of the host processor. Packet filtering then occurs in two successive stages beginning at the controller, which examines packets in real-time. To accomplish this, the controller is "conditioned" with an appropriate subset of the specified filtering criteria, according to the filtering capabilities of that controller. The controller classifies packets into three categories: Those not satisfying the filter criteria ("rejects"); those satisfying the criteria ("exact matches"); and those possibly satisfying the criteria ("partial matches"). Rejects are not delivered to the processor. Those packets which are classified as exact or as possible matches are delivered, with appropriate indications of their classification, to the device processor. The controller, ideally, excludes as many unwanted packets as its capabilities will allow, and the host processor (with the appropriate software operating therein) completes the overall filtering operation, as required. The value of filtering packets at the controller level (the partial filtering) is that it reduces the burden on the host processor.

Controller filtering implementations are constrained by the fact that they must process packets in real-time with packet reception. This places a high value on filtering mechanisms that can be implemented with a minimum amount of logic and memory. Controller based filtering criteria are contained in a target memory. In the case of exact matching, a literal list of desired targets is stored in the target memory. While exact matching provides essentially perfect filtering, it can be used in applications wherein there are only a very small number of targets.

Partial filtering is employed when the potential number of targets is relatively large, such as is often the case in multicast applications. A primary consideration is the "efficiency" of the partial filter. Efficiency (E), in this context, may be expressed as:

$$E=Tn/Pn$$

where:

Tn=the number of target packets of interest; and

Pn=the number of potential candidates delivered to the processor.

An efficiency of E=1.0 represents an exact filtering efficiency wherein every candidate is a desired target. This is the efficiency of the filtering which occurs in the "exact matching" previously discussed herein. While exact filtering efficiency is an objective, the previously mentioned constraints, including that the controller must do its filtering in essentially real-time, will generally not allow for such efficiency.

The predominant method used in the prior art for partial packet filtering is "hashing". The process conventionally begins with the extraction from each received packet of all fields involved in the specified filtering criteria. The composite of such relevant fields is called the "candidate field". Assuming an even distribution of candidate fields (a situation that is not always literally accurate, but the assumption of which is useful for purposes of analysis), there will be a potential number of packet candidates of 2^{Cb} where Cb is the number of bits in the candidate field. The hashing function produces a reduction in the bit size of the candidate field according to a "hashing function". As a part of the initiation of the controller, the hashing function is applied to each field of the target memory to assign a "target hash value" to each such field. The controller memory is initialized as a bit mask representing the set of target hash values. Then, during operation, a "candidate hash value" is created by applying the hashing function to each candidate field. The candidate hash value is used as a bit index into the controller memory, with a match indicating a possible candidate.

As can be appreciated in light of the above discussion and from a general understanding of simple hashing operations, the hashing function has the effect of partitioning the 2^{Cb} candidate possibilities into Mb groups (called "buckets"), where Mb is the number of bits in the controller's target memory. Because candidate packets that fall into the same bucket are not distinguished, a "hit" represents any of 2^{Cb}/Mb candidates. Useful hashing functions will partition the candidate possibilities in a roughly uniform distribution across the set of Mb buckets. For a single target, the efficiency of such a hashing method is Mb/2^{Cb} If Tn desired targets are represented by Bn buckets (where Bn<=Tn and Bn<=Mb, the efficiency of such a hashing method is:

$$E=Tn/(Bn2^{Cb}/Mb)=TnMb/Bn2^{Cb}$$

In exact matching, target memory could hold Mb/Cb targets. Hashing is appropriate when the number of buckets (Bn) is larger than this figure. However, effective hashing also requires that the number of buckets be less than Mb, because as target memory density increases there is less differentiation among candidate fields. With the target memory full of hash targets, Bn=Mb and the efficiency is Tn/2^{Cb}.

As can be appreciated, the described prior art hashing method used for partial packet filtering implies a loss of information in that a single hash value potentially represents a large set of candidates. Clearly, it would be desirable to reduce such loss of data. Correspondingly, it would desirable to maximize the filtering efficiency for a given Mb or (or to minimize the Mb for a given filter efficiency).

To the inventor's knowledge, no prior art method for partial packet filtering has improved efficiency or reduced data loss as compared to the conventional hashing method described above.

DISCLOSURE OF INVENTION

Accordingly, it is an object of the present invention to provide a method and means for efficiently performing a partial filtering operation on data packets in a computer network.

It is another object of the present invention to provide a method and means for partial packet filtering which rejects a maximum number of incoming packets which are not at interest without requiring a large target memory and without unduly slowing down the processing of incoming packets.

It is still another object of the present invention to provide a partial packet filtering method and means which is inexpensive to implement.

It is yet another object of the present invention to provide a partial packet filtering method and means which will operate in real-time or near real-time.

It is still another object of the present invention to provide a partial packet filtering method and means which is adaptable to a variety of network system requirements.

Briefly, the preferred embodiment of the present invention implements multiple independent hashing functions applied in parallel to the candidate field of each packet. The combined application of multiple independent hashing functions results in specification of a hash matrix, with each coordinate of the hash matrix being the result of one of the hashing functions. The hash matrix includes the results of different hashing algorithms applied to a single candidate field, or the same hashing function applied to different subsets of the candidate field, or a combination thereof. The filter parameters consist of the set of acceptable result values for each hashing operation.

An advantage of the present invention is that partial packet filtering efficiency is improved, thereby freeing the host processor from a substantial portion of the packet filtering operation.

Yet another advantage of the present invention is that filtering efficiency is increased geometrically with an increase in target memory.

Still another advantage of the present invention is that a minimum amount of target memory is required for a specific target efficiency.

Yet another advantage of the present invention is that the partial packet filtering can be performed in a minimum amount of time for a given target efficiency.

These and other objects and advantages of the present invention will become clear to those skilled in the art in view of the description of the best presently known modes of carrying out the invention and the industrial applicability of the preferred embodiments as described herein and as illustrated in the several figures of the drawing.

BRIEF DESCRIPTION OF THE DRAWING

FIG. 1 is a block diagram depicting a portion of a computer network with an improved partial packet filter according to the present invention in place therein;

FIG. 2 is a diagrammatic representation of a conventional prior art Ethernet data packet;

FIG. 3 is diagrammatic representation of a hash table;

FIG. 4 is a flow chart showing a conventional prior art partial packet filtering operation;

FIG. 5 is a block depiction of a partial packet filtering method according to the present invention;

FIG. 6 is a flow chart, similar to the chart of FIG. 4, depicting the packet processing operation series of FIG. 5; and

FIG. 7 is a flow chart depicting the preliminary operation series of FIG. 5.

BEST MODE FOR CARRYING OUT INVENTION

The best presently known mode for carrying out the invention is a partial packet filter for implementation in a personal computer resident Ethernet controller. The predominant expected usage of the inventive improved packet filter is in the interconnection of computer devices, particularly in network environments where there are relatively few targets.

The improved partial packet filter of the presently preferred embodiment of the present invention is illustrated in a block diagram in FIG. 1 and is designated therein by the reference character 10. In the diagram of FIG. 1, the improved partial packet filter 10 is shown configured as part of a network system 12 (only a portion of which is shown in the view of FIG. 1). In many respects, the best presently known embodiment 10 of the present invention is structurally not unlike conventional partial packet filter mechanisms. Like prior art conventional partial packet filters, the best presently known embodiment 10 of the present invention has a controller 14 with an associated target memory 16. In the example of FIG. 1, the improved partial packet filter 10 receives data from a network node 18 and performs the inventive improved packet filtering process on such data before passing selected portions of the data on to a host processor 18 to-which-the-improved-partial-packet filter 10 is dedicated.

FIG. 2 is a diagrammatic representation of a conventional Ethernet data packet 210. The standardized Ethernet packet 210 has a preamble 212 which is 64 bits in length, a destination address 214 which is 48 bits in length, a source address 216 which is 48 bits in length, a length/type field 218 which is 16 bits in length and a data field 220 which is variable in length from a minimum of 46 eight bit bytes to a maximum of 1500 bytes. Following the data field 220 in the packet 210 is a 4 byte (32 bit) frame sequence check ("FCS") 222. The packet 210 is transmitted serially beginning at a "head" 224 and ending at a "tail" 226 thereof. The preamble 212, destination address 214, source address 216 and length/type field 218 are collectively referred to as the header 219.

FIG. 3 is a diagrammatic representation of a conventional single dimensional hash table 310 with which one skilled in the art will be familiar. The hash table 310 has a plurality of address locations 312 each of which can be "set" (set to 1) or left unset (set to zero).

FIG. 4 is a flow diagram depicting the operation of a conventional prior art partial packet filtering operation 410. As previously discussed briefly, a packet 210 (FIG. 2) is received (receive packet operation 412) from the network 18 (FIG. 1) and a candidate field 413 (such as the header 219 of the packet 210) is extracted (extract candidate field operation 414). A hashing operation 416 is performed on the extracted candidate field 413 to produce a hash value 417 and the hash value 417 is compared to the hash table 310 (FIG. 3) stored in the target memory 16 (FIG. 1) in a comparison operation 418. If the result of the comparison operation 418 is a match, the packet 210 is forwarded in a forward packet operation 420. If the result of the comparison operation 418 is not a match, the packet 210 is rejected 422 in a reject packet operation. It should be remembered that the use of the header 219 here is an example only, and any portion or combined portions of the packet 210 might constitute the candidate field 413 in a given application.

FIG. 5 is a flow diagram depicting the inventive improved packet filtering process 510. The improved packet filtering process 510 is accomplished in a preliminary operation series 512 and a packet processing operation 514, each of which is repeated as required, as will be discussed hereinafter. The preliminary operation series 512 is accomplished according to software residing in the host processor 20 (FIG. 1) to configure the target memory 16 (FIG. 1) as will be discussed hereinafter. It should be noted that the fact that the improved packet filtering process 510 is divided into the two main operation categories (the preliminary operation series 512 and the packet processing operation 514) does not distinguish this invention over the prior art. Rather, the processes within the preliminary operation series 512 and the packet processing operation 514 describe the essence of the inventive process.

FIG. 6 is a flow chart showing the inventive packet processing operation 514 in a manner analogous to the presentation of the prior art partial packet filtering operation 410 depicted in FIG. 4. As can be seen in the view of FIG. 6, the packet processing operation series 514 is similar in many respects to the prior art partial packet filtering process 410 (FIG. 4). In the packet processing operation series 514, a packet 210 (FIG. 2) is received (receive packet operation 412) and a candidate field 413 is extracted in an extract candidate field operation 414. In the best presently known embodiment 10 of the present invention, the inventive packet processing operation series 514 next performs a candidate field reduction operation 626. In the best presently known embodiment 10 of the present invention, the candidate field reduction operation 626 is merely the application of the conventional CRC polynomial algorithm to the candidate field 413 to yield a 32 bit CRC output value 628 (although any of a number of similar algorithms might be applied for this purpose). Next, a subset selection operation 630 selects a predetermined number (two in the example of FIG. 6) of bit-wise subsets 636 from the CRC output value 628. The method for determining the quantity of bit-wise subsets 636 to be selected in the subset selection operation 630, and the size of each, will be discussed hereinafter. In the best presently known embodiment 10 of the present invention, the bit-wise subsets 636 are each 6 bits in length. It should be noted that, in the best presently known embodiment 10 of the present invention, the bit-wise subsets 636 are selected from the CRC output value 628 simply by taking the first 6 bits of the CRC output value 628, the second six bits, and so on until as many bit-wise subsets as are needed are obtained and so, in the best presently known embodiment 10 of the present invention, the bit wise subset 636 are "consecutive bit section " of the fixed size field (the CRC output value 628 in the best presently known embodiment 10 of the present invention. The inventors have determined that the bits of the CRC output value 628 (resulting from the CRC polynomial function) are independent of each other, and so any 6 bit portion of the CRC output value 628 is as representative of the CRC output value 628 as is any other 6 bit portion.

The bit-wise subsets 636 are then compared to the hash table 310 (FIG. 3) stored in the target memory 16 (FIG. 1) in a comparison operation 642. The combined multiple hash values 636 may be considered to be a hash matrix 638 (in the example of FIG. 6, a two dimensional hash matrix 638).

It is important to note that the essence of the present inventive method lies in the extraction of the plurality of independent or relatively independent representative indices of the candidate field 413 ("candidate filled indices") which, in the example of the best presently known embodiment 10 of the present invention are the bit-wise subsets 636 which make up the hash matrix 638. That is, the bit-wise subsets 636 are representative fields in that the bit-wise subsets 636 are representative of the candidate field 413, as discussed above. The generally simultaneous (parallel) processing of these is the source of the advantages of the present inventive method and means. The exact method described herein in relation to the best presently known embodiment 10 of the present invention, that of first reducing the candidate field 413 in the candidate field reduction operation 626 and then extracting the bit-wise subsets 636 is but one of many potential methods for accomplishing such a parallel hashing operation 639, and the present invention is not intended to be limited by this aspect of the best presently known embodiment 10.

In the best presently known embodiment 10 of the present invention, in a comparison operation 642, each of the bit-wise subsets 636 is compared to a reference hash table 644 (a "target hash array") stored in the target memory 16 (FIG. 1) and only if all match is the packet 210 forwarded in a packet forwarding operation 646. In the example of FIG. 6, the reference hash table 644 will be a 64 element array representing all values from 0 through 63 inclusive. Some elements of the reference hash table 644 are set as will be discussed hereinafter in relation to the preliminary operation series 512. If the value of the bit-wise subset "falls into one of the buckets" (is equivalent to a corresponding set bit in the reference hash table **644**), then the data packet **210** is defined as being a "match".

Now returning to a consideration of the preliminary operation series **512** (FIG. **5**) with an understanding of the packet processing operation series **514**, the target memory **16** is configured in process steps much like those described in relation to the packet processing operation series **514** of FIG. **6**.

FIG. **7** is a flow diagram of the preliminary operation series **512** according to the best presently known embodiment **10** of the present invention. A preliminary operation which is common to both the prior art and the present invention is a target field(s) selection process **712**. The target (field) s selection process is merely the selection of criteria to which incoming packets **210** are to be compared. For example, if the entire process is to be on the basis of desired destinations, then an intended destination address **214** (FIG. **2**) will be (one of) the target field(s) **714**, and if three destinations are of interest, then there will be three target fields **714** as illustrated in the example of FIG. **7**. The actual process involved in selecting the target field(s) is a function of network control software which is found in the prior art and which is not relevant to the present invention except to the extent that it delivers the target field(s) **714** to the inventive preliminary operation series **512**.

Having determined the quantity of target fields **714** of interest, host software will next determine a bit-wise subset quantity **716** (the appropriate "subset quantity" of bit-wise subset **636**) in a bit-wise subset quantity determination operation **718**. The bit-wise subset quantity determination operation **718** will be discussed in more detail hereinafter, as it can be better understood in light of the present description of the entire preliminary operation series **512**. For the present simplified example of FIGS. **6** and **7**, and as already mentioned, the bit-wise subset quantity **716** is two. That is, two of the bit-wise subsets **636** are to be extracted from the CRC output value **628** in the subset selection operation **630** of FIG. **6**.

As can be appreciated, the target fields **714** are each equivalent in form to the candidate fields **413** discussed previously herein, and processing of the target fields **714** is much the same as has been previously described herein in relation to the candidate fields **413**. In the inventive preliminary operation series **512**, each of the target fields **714** is processed in a target field reduction operation **726** by application of the CRC polynomial to produce a target CRC value **728**. Each of the target CRC values **728** is then processed in a target subset selection operation **730** to produce a plurality (two for each target CRC value **728** for a total of six, in the present example) of target bit-wise subsets **736**. In more general terms, each of the "target fields **714** (having been selected according to prior art methods as discussed previously, herein) is processed as described to produce a "target representative field" (the target CRC value **728** in the present example), which is then further processed as described to produce the "target indices", which target indices may be "target string subsets**38** of the target representative field and which are, in the present example, the target bit-wise subsets **736**. This process is alike to the process which is repeated as necessary to process each incoming data packet **210**, wherein the candidate fields **413** are processed to produce a candidate representative field (the CRC output value **628** in the present example), which is further processed to produce the "candidate string subsets" (the bit-wise subsets **636** in the present example). The quantity of target bit-wise subsets **736** taken from each target CRC value **728** is also the bit-wise subset quantity **716** (two, in the present example). It should be noted that a target parallel hashing operation **739** is like the previously described parallel hashing operation **639** in that the invention might be practiced with variations of the specific steps therein which are presented here as features of the best presently known embodiment **10** of the present invention.

In a target memory setting operation **740** the reference hash table **644** is formatted such that each memory location **312** corresponding to a value of any of the target bit-wise subsets **736** is set. For example, if the first target bit-wise subset **736a** were "000010" (decimal value 2) then the third memory location **312c** in the reference hash table **644** would be set to "1", as is illustrated in FIG. **7**. As can be appreciated from the above discussion, the maximum number of memory locations **312** in the reference hash table **644** which can be set by this process is the quantity of target bit-wise subsets **736** (six, in the present example). However, since two or more of the target bit-wise subsets might coincidentally hash to the same value, a lesser quantity of memory locations **312** might also be set.

Now returning to a more detailed discussion of the bit-wise subset quantity determination operation **718**, the target memory **16** is to be configured to maximize the effectiveness of the filtering based on the quantity of multicast packets **210** of interest to the software of the host processor Therefore, the bit-wise subset quantity determination operation **718** attempts to determine (or, at least, to approximate) an optimal number of indices per packet (and, thus, the bit-wise subset quantity **716** discussed previously herein). The "optimal" number here means that which will minimize the number of "uninteresting" packets which match the set data bits **312** in the reference hash table **644** while matching all of the "interesting" packets **210**. In the best presently known embodiment **10** of the present invention, the following table is used to determine the bit-wise subset quantity **716**.

TABLE OF SUBSET QUANTITIES

Addresses of Interest	Number of Hash Indices Bit-Wise Subset Quantity 716
1–2	5
3	4
4–9	3
10–16	2
17 or more	1

The above table is offered here as a guide only, in that the "optimal" number of selected hash indices may vary in ways not presently contemplated. Furthermore, it should be noted that the above table is based upon an assumption that none of the target indices (the target bitwise subsets **736** in the best presently known embodiment **10** of the present invention hash to the same memory locations **312** in the reference hash table **644**. If, indeed, two or more of the target bit-wise subsets **736** did hash to the same memory location **312**, then additional hash indices could be added to increase efficiency without sacrificing speed or requiring additional memory or processing.

It should be noted that while the packet processing operation series **514** is accomplished in the hardware of the best presently known embodiment **10** of the present invention, the preliminary operation series (which can be accomplished at a more leisurely pace) is performed primarily by software of the host processor **20**. As can be appreciated in light of the above discussion, the preliminary operation series will be repeated when the network **12** is reconfigured, when it is desired to communicate with additional members of the network **12**, or upon other occasions according to the needs of the user and the network **12**. The packet processing operation series **514** will be repeated whenever an incoming packet is detected from the network node **18**.

It should also be noted that, while the best presently known embodiment **10** of the present invention hashes each of the CRC values **628** and **728** to a common reference hash table **644**, the invention might be practiced with equal efficiency by hashing each of the CRC values **628** and **728** to its own individual hash table (not shown). Using the quantities of the example of FIGS. **6** and **7**, each of the individual hash tables would be 32 bits (memory locations **312**) large (one half of 64 bits, since it must be divided between the two target CRC values **728**). The individual bit-wise subsets **636** and **736** would then be 5 bits long (decimal value 0 through 31).

Various modifications may be made to the inventive improved packet filter **10** without altering its value or scope. For example, the quantity, size, and derivation of the plurality of bit-wise subsets **636** and **738** could readily be revised according to the parameters discussed herein.

All of the above are only some of the examples of available embodiments of the present invention. Those skilled in the art will readily observe that numerous other modifications and alterations may be made without departing from the spirit and scope of the invention. Accordingly, the above disclosure is not intended as limiting and the appended claims are to be interpreted as encompassing the entire scope of the invention.

INDUSTRIAL APPLICABILITY

The improved partial packet filter **10** is adapted to be widely used in computer network communications. The predominant current usages are for the interconnection of computers and computer peripheral devices within networks and for the interconnection of several computer networks.

The improved partial packet filters **10** of the present invention may be utilized in any application wherein conventional computer interconnection devices are used. A significant area of improvement is in the inclusion of the parallel processing of a plurality of indices (bit-wise subsets **636**) of a packet

The efficiency of the filtering provided by the improved partial packet filter **10** is significantly improved, particularly for cases where the number of targets is small relative to the number of "buckets" (memory locations **312**). To compare the efficiency of the present inventive improved packet filtering process **510** embodied in the improved partial packet filter **10** with the prior art partial packet filtering process **410**, assume, for example, the following values:

Mb=64 (representing 64 memory locations **312** in the reference hash table **644**)

Cb=48 (representing a 48 bit candidate field **413** size—a typical size of the destination address **214**

Dn=4 (representing a bit-wise subset quantity **716** of four)

Then, the prior art partial packet filtering process **410** will partition the 2^{Cb} possibilities among 64 distinct buckets, one of which matches the bucket into which the single target falls. In the improved packet filtering process **510**, the four parallel hashing functions partition among 16 possible buckets each. The efficiency (Ef) for the prior art partial packet filtering process **410** would then be:

$$Ef=64/2^{48}=\frac{1}{2}^{42}$$

The efficiency (Ef4) for this example of the improved packet filtering process **510** is:

$$Ef4=64^4/(4^4*2^{48})=\frac{1}{2}^{32}$$

The efficiency Ef4 is better than the efficiency Ef by a factor of 2^{10} (1024), which is to say that only a thousandth as many (uninteresting) packets will be delivered to the next stage of filtering using the inventive improved partial packet filter **10** as compared to the prior art.

Filtering of packets may be accomplished through a combination of exact and partial match filters. Typically, one or more partial filterings will occur first, with the multiple dimensions of each filtering accomplished in parallel with each other (according to the present invention). Packets which pass through the inventive improved partial packet filter **10** may then be filtered using an exact match filter technique, such as "binary search lookup" of the filter data in a sorted table of acceptable filter data values. Furthermore, results of partial filtering can be used to determine which of many (possibly sorted) tables in which to search for the packet.

Accordingly, the inventive improved packet filtering process **510** may be applied more than once to each incoming packet **210** (in a first stage and a second stage). In such an example, configuration of the first stage partial filtering would involve specification of the number and type of hashing operations to be performed, along with the portion of the packet which is to comprise the filter data for each such operation, along with acceptable results for each. Multiple partial filterings may be configured with the specification including the logical relation to apply to the results of each filtering. For example, partial filtering A might be to apply the 32 bit CRC polynomial to the destination address field of an Ethernet packet, and retain the lowest order 3 bits—a value from 0 to 7. Partial filtering B might be to apply the 32 bit CRC polynomial to the source address field of the Ethernet packet, and retain the lowest order 3 bits. The logical relation might be to accept packets only for which the results of the first filtering (A) is either 2 or 4, and the result of the second filtering (B) is either a 3 or a 4. In a general case, one may expect the likelihood of arbitrarily filter data to "pass" the first filtering to be 2 in 8 (25%), since 2 of the 8 values from 0 to 7 are acceptable. Similarly, the likelihood of the second filtering "passing" such a filter is 2 in 8 (25%). Assuming that the two filterings are, as desired, truly independent, the likelihood of this arbitrary packet being accepted is the product of these, or 1 in 16. Note further that the specification of these "acceptable result sets" ({2,4} for A and {3,4} for B) requires 16 bits of information for full specification, where 8 bits indicate the acceptability/unacceptability of each of the 8 possible values of filtering A, and 8 additional bits indicate the acceptability/unacceptability of each of the 8 possible values of filtering B. Use of such multiple partial filterings may be especially effective in situations where filtering criteria are derived from independent portions of the filter data, such as filtering for all packets whose destination address OR whose source address is within a set of interesting addresses AND whose packet type indicates a particular protocol of interest.

Since the improved partial packet filters of the present invention may be readily constructed and are compatible with existing computer equipment it is expected that they will be acceptable in the industry as substitutes for conventional means and methods presently employed for partial packet filtering. For these and other reasons, it is expected that the utility and industrial applicability of the invention will be both significant in scope and long-lasting in duration.

We claim:

1. A method for selectively forwarding a data packet and controlling the distribution of data packets in a computer network system, the data packet having a candidate field containing information about the data packet, the method comprising:

configuring a target memory of a controller to contain a target hash array in steps including;

aa determining a target field and extracting a plurality of target indices from said target field, the target indices being a binary number having a value;

ab setting memory locations in the target memory corresponding to the value of each of the target indices; and

processing the data packet in steps including:

ba extracting the candidate field from the data packet;

bb extracting from the candidate field a plurality of candidate field indices;

bc comparing the values of each of the candidate field indices to the target hash array; and

bd forwarding the packet when each of the values of each of the candidate field indices corresponds to a memory location of the target hash array which was set in step ab.

2. The method of claim 1, wherein:

step aa is accomplished in substeps including:

aa1 reducing the target fields to a plurality of target representative fields; and

aa2 selecting one or more target string subsets from the target representative field; and

step bb is accomplished in substeps including:

bb1 reducing the candidate field to a plurality of candidate representative fields; and

bb2 selecting one or more candidate string subsets from the target representative field.

3. The method of claim 1, wherein:

step ab is accomplished by causing only those memory locations in the target memory which correspond to the value of each of the target string subsets to contain a value of one.

4. The method of claim 2, wherein:

step aa1 is accomplished by applying a cyclic redundancy check algorithm to each of the target fields; and

step bb1 is accomplished by applying the same cyclic redundancy check algorithm to the candidate field.

5. The method of claim 2, wherein:

in step aa2 the target string subsets are selected by extracting a plurality of target bit-wise subsets from the target representative field; and

in step bb2 the candidate string subsets are selected by extracting a plurality of candidate bit-wise subsets from the representative candidate field.

6. The method of claim 2, and further including:

an additional process step preceding step ab wherein a subset quantity is determined, the subset quantity being the number of target string subsets to be extracted from each of the target representative fields and also the number of candidate string subsets to be extracted from each of the candidate representative fields.

7. The method of claim 6, wherein:

the additional process step is accomplished, at least initially, by selecting the subset quantity from a table of subset quantities.

8. The method of claim 2, wherein:

each of the target representative target and the candidate representative field are 32 bits in length.

9. The method of claim 1, wherein:

steps aa through ab are repeated when a change in the distribution of data packets is desired.

10. The method of claim 1, wherein:

steps ba through bd are repeated for each incoming data packet.

11. The method of claim 1, and further including:

an additional process step preceding step ab wherein a subset quantity is determined, the subset quantity being the number of target indices to be extracted from each of the target fields and also the number of candidate indices to be extracted from each of the candidate fields.

12. The method of claim 11, wherein:

the additional process step is accomplished, at least initially, by selecting the subset quantity from a table of subset quantities appropriate to a quantity of target quantities.

13. The method of claim 1, wherein:

the candidate field includes a target address field of the data packet.

14. The method of claim 1, wherein:

the data packet is a standardized Ethernet data packet.

15. The method of claim 1, wherein:

the target hash array is an unapportioned array such that each of the target indices is used to set memory locations in that unapportioned array.

16. The method of claim 1, wherein:

the target hash array is apportioned such that at least some of the target indices are directed to different portions of the target hash array.

17. The method of claim 1, wherein:

the target indices and the candidate indices are each a binary string of fixed bit length.

* * * * *

+ TCAM

US005841874A

United States Patent [19]

Kempke et al.

[11] **Patent Number: 5,841,874**

[45] **Date of Patent: Nov. 24, 1998**

[54] **TERNARY CAM MEMORY ARCHITECTURE AND METHODOLOGY**

[75] Inventors: **Robert Alan Kempke**, Tempe, Ariz.; **Anthony J. McAuley**, Bloomfield, N.J.

[73] Assignee: **Motorola, Inc.**, Schaumburg, Ill.

[21] Appl. No.: **696,453**

[22] Filed: **Aug. 13, 1996**

[51] **Int. Cl.**^{6} **H04L 9/00**

[52] **U.S. Cl.** **380/50**; 380/4; 380/9; 380/23; 380/25; 380/49; 380/59; 326/59; 365/168

[58] **Field of Search** 380/4, 9, 23, 25, 380/28, 49, 50, 59; 326/59; 341/57; 365/168; 371/37.04; 395/427, 428, 435

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,599,205	8/1971	Cornelis et al.	341/57
3,671,959	6/1972	Amano	341/57
3,706,977	12/1972	Dailey et al. .	
3,760,277	9/1973	Whang	341/57 X
3,798,544	3/1974	Norman	380/9 X
4,231,023	10/1980	Warner	341/57
4,296,475	10/1981	Nederlof et al. .	
4,809,224	2/1989	Suzuki et al.	365/168
4,910,750	3/1990	Fisher	341/57 X
5,432,735	7/1995	Parks et al.	365/168
5,498,980	3/1996	Bowles	326/59 X

FOREIGN PATENT DOCUMENTS

650167A	10/1994	European Pat. Off. .

OTHER PUBLICATIONS

An article entitled "Design of a Key Agile Cryptographic System for OC-12c Rate ATM" by Daniel Stevenson, Nathan Hillery, Greg Byrd, Fengmin Gong and Dan Winkelstein, from 1995 IEEE.

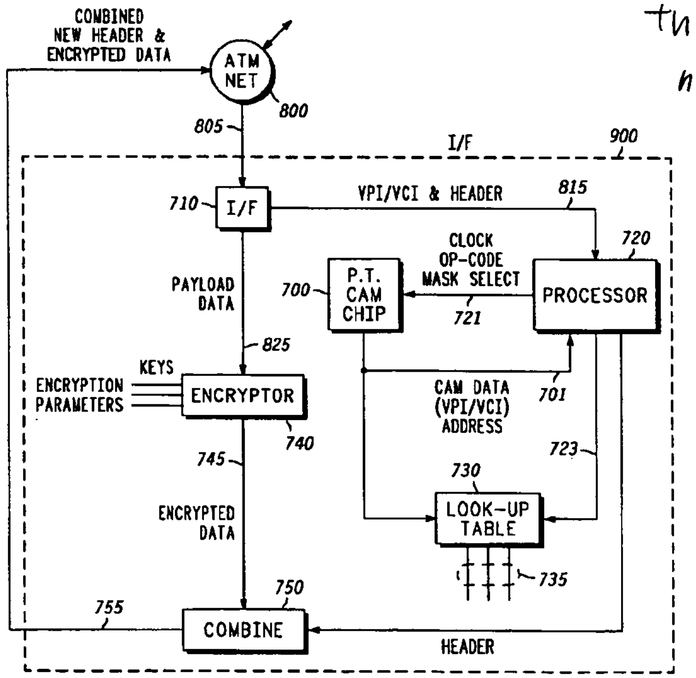
An article entitled "A Design Of A High-Density Multi--Level Matching Array Chip For Associative Processing", IEICE Transactions, vol. E. 74, No. 4 Apr. 1991.

Primary Examiner—Bernarr E. Gregory
Attorney, Agent, or Firm—James A. Coffing; Bradley J. Botsch; John C. Scott

[57] **ABSTRACT**

The present invention encompasses a method of storing ternary data that includes the steps of (1) initializing a conversion register by storing binary-to-ternary mask data in a conversion register; (2) storing ternary data in a content addressable memory (CAM) by inputting a single bit binary data to the conversion register, and converting the binary data into two bits of ternary data using the conversion register; and (3) simultaneously storing the two bits of ternary data in first and second memory cells. For subsequent searching, the method further includes the steps of searching for a match of input search binary data to the stored contents of the CAM; providing a match valid output responsive to the input search binary bits matching any of the stored contents; and generating an address corresponding to a location in the CAM where the match is found.

21 Claims, 9 Drawing Sheets



this uses ATM –
not combination w/ cache
and show
encryption key
look up
no destination
routing to
devices
clearly
shown

06/25/2003, EAST Version: 1.03.0002

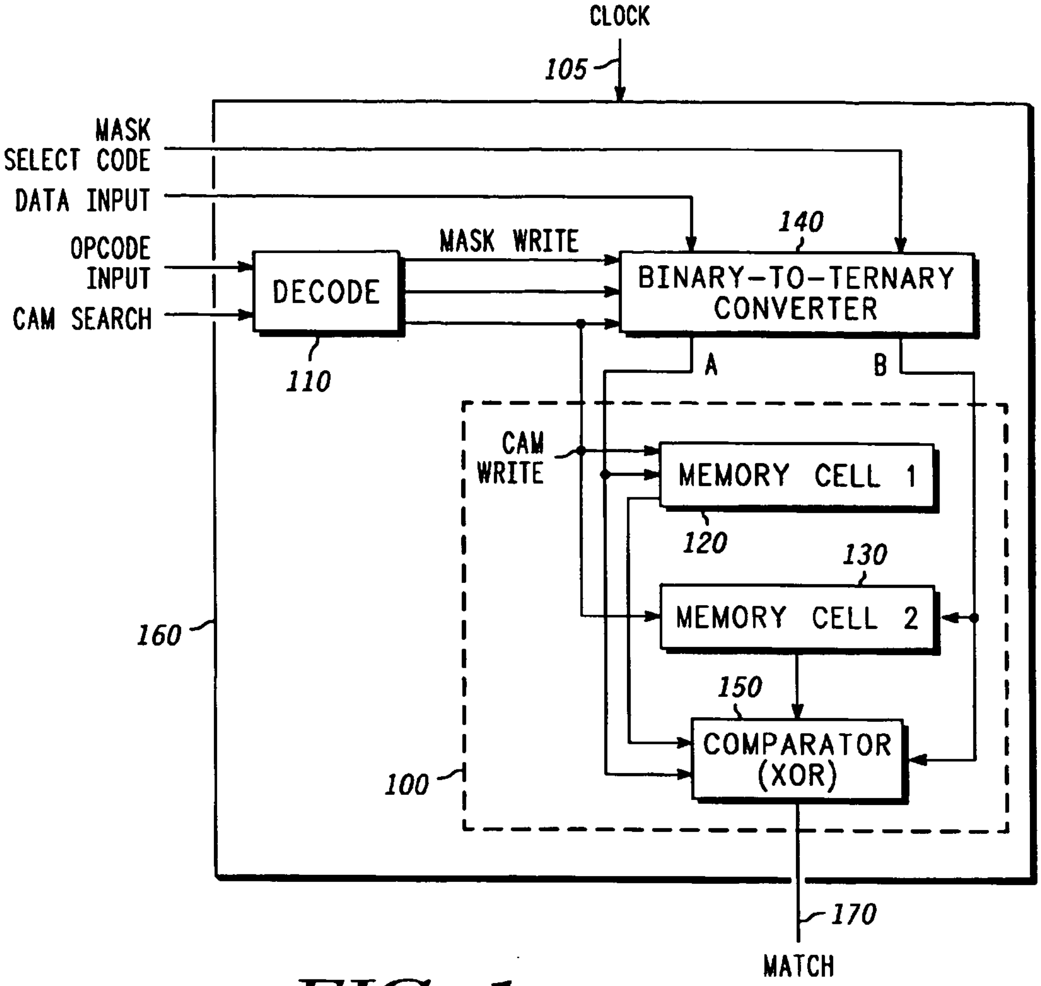

CLOCK
105
MASK SELECT CODE
DATA INPUT
OPCODE INPUT
CAM SEARCH
DECODE
110
MASK WRITE
140
BINARY-TO-TERNARY CONVERTER
A
B
CAM WRITE
MEMORY CELL 1
120
130
MEMORY CELL 2
150
COMPARATOR (XOR)
160
100
170
MATCH

FIG. 1

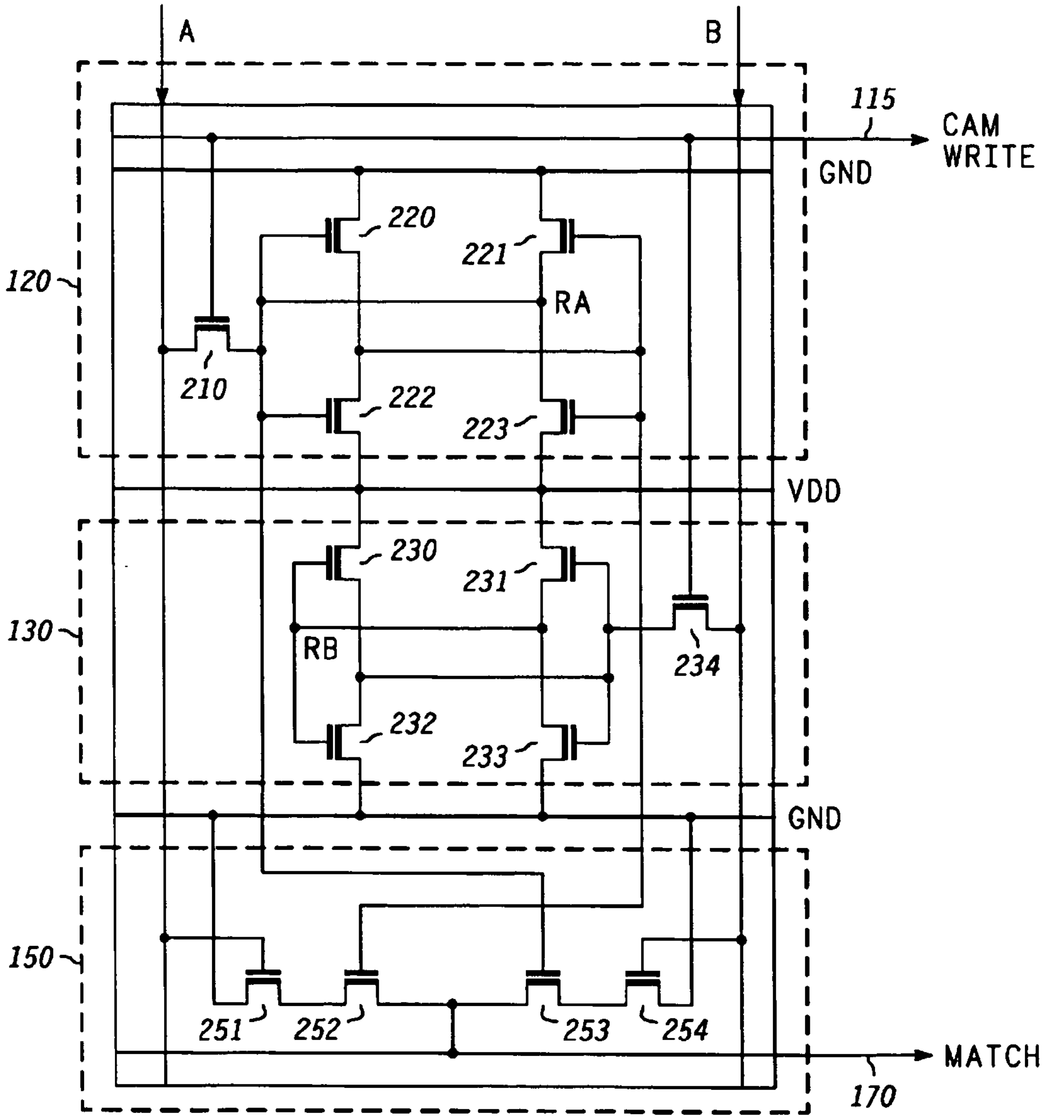
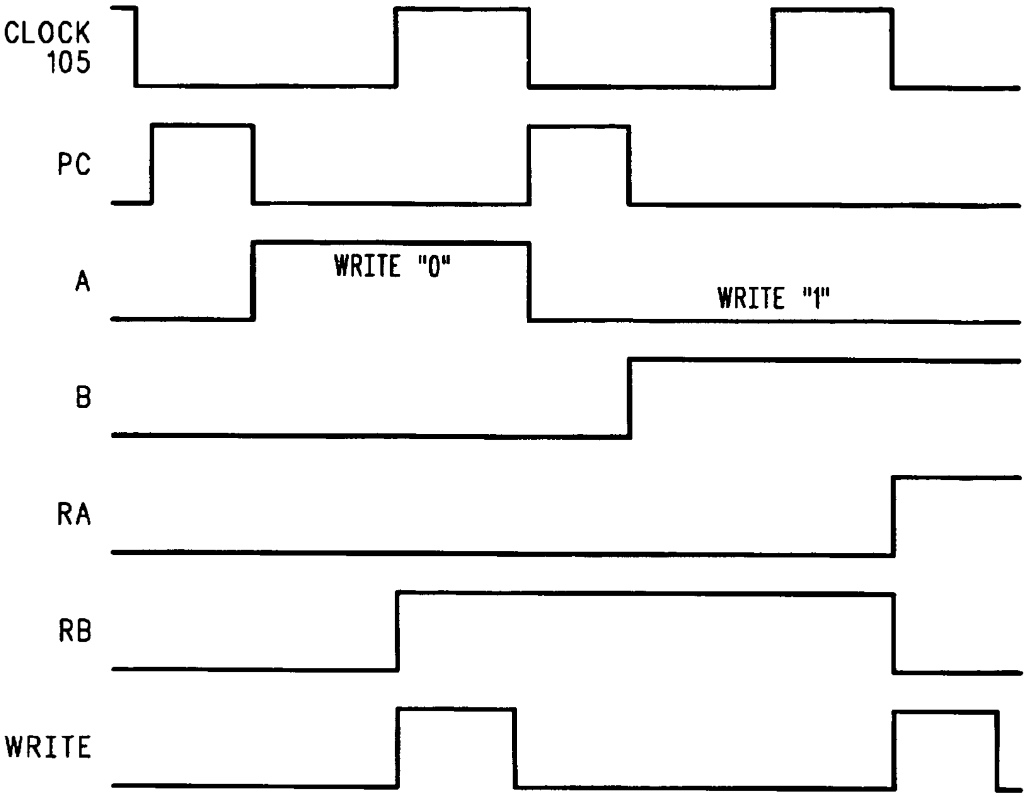
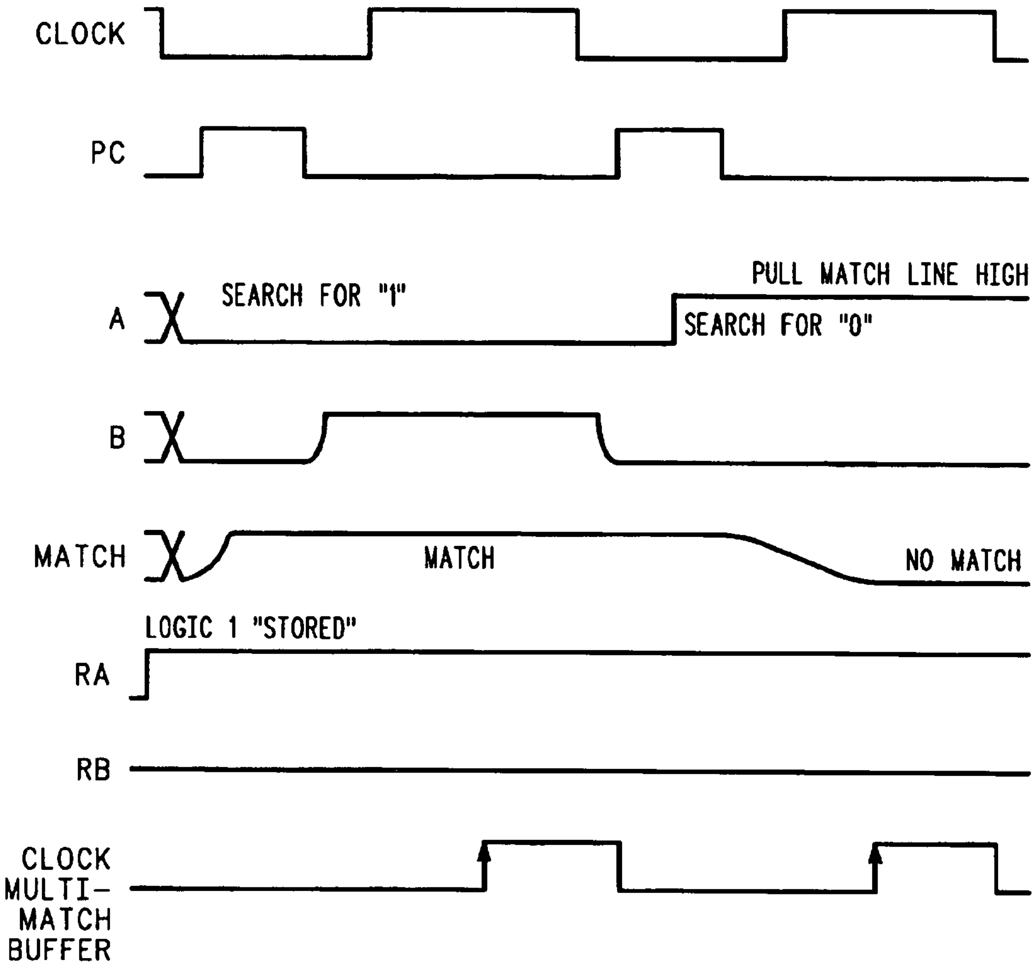

A
B
115
CAM WRITE
GND
120
220
221
RA
210
222
223
VDD
130
230
231
RB
234
232
233
GND
150
251
252
253
254
MATCH
170

FIG. 2



WRITE CYCLE

FIG. 3



SEARCH CYCLE

FIG. 4

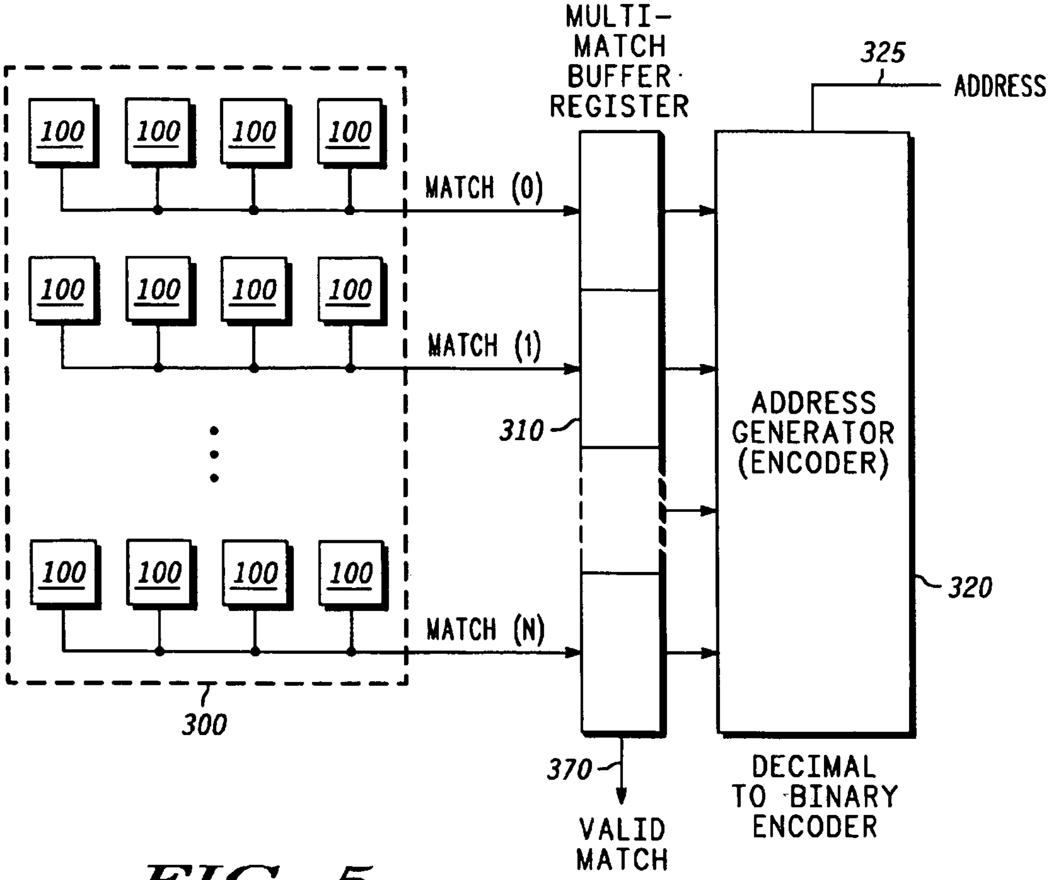

MULTI-
MATCH
BUFFER
REGISTER
325
ADDRESS
100
MATCH (0)
MATCH (1)
310
ADDRESS
GENERATOR
(ENCODER)
320
MATCH (N)
300
370
VALID
MATCH
DECIMAL
TO BINARY
ENCODER

FIG. 5

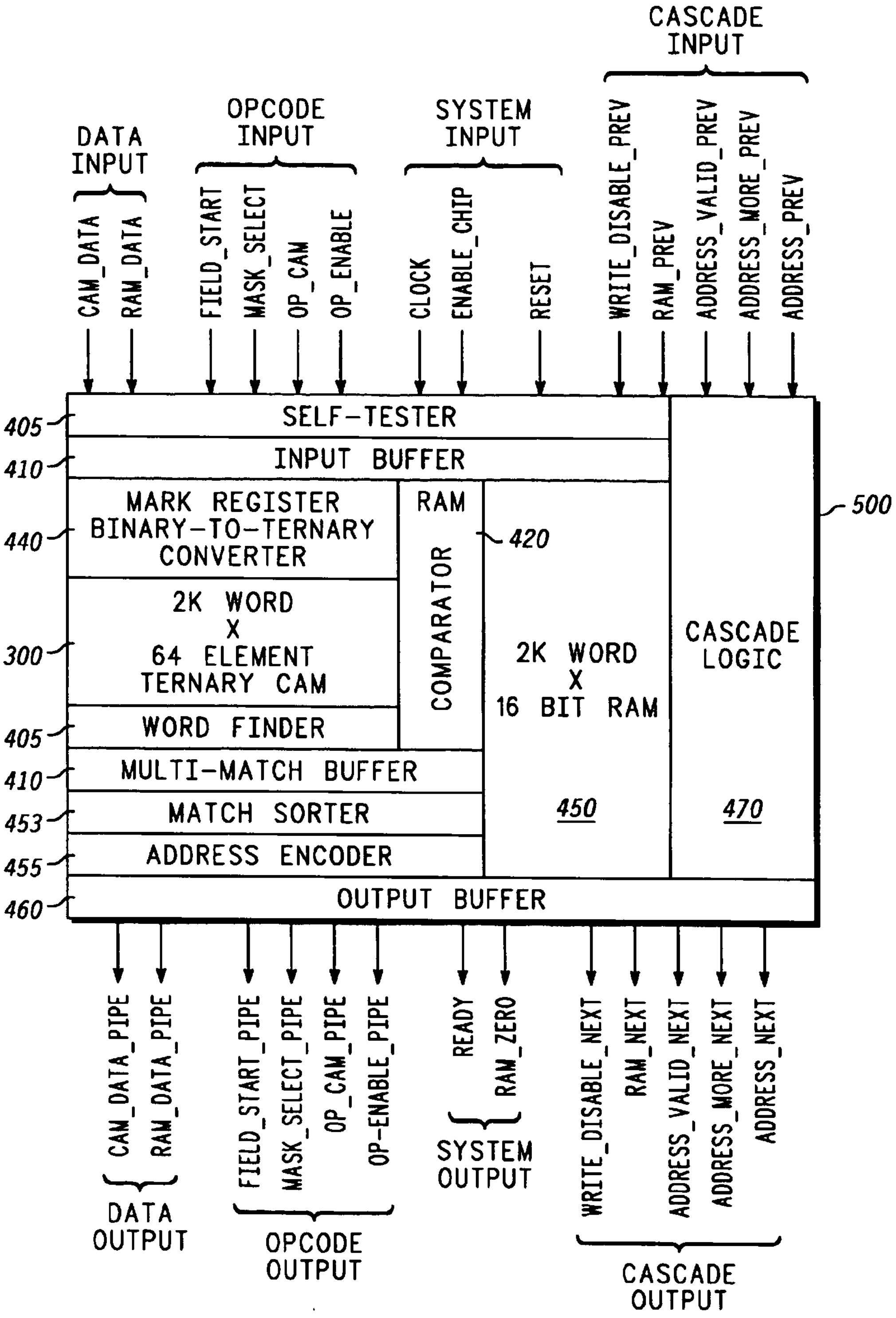

CASCADE INPUT
DATA INPUT
OPCODE INPUT
SYSTEM INPUT
CAM_DATA
RAM_DATA
FIELD_START
MASK_SELECT
OP_CAM
OP_ENABLE
CLOCK
ENABLE_CHIP
RESET
WRITE_DISABLE_PREV
RAM_PREV
ADDRESS_VALID_PREV
ADDRESS_MORE_PREV
ADDRESS_PREV
405
SELF-TESTER
410
INPUT BUFFER
440
MARK REGISTER BINARY-TO-TERNARY CONVERTER
RAM COMPARATOR
420
500
300
2K WORD X 64 ELEMENT TERNARY CAM
2K WORD X 16 BIT RAM
CASCADE LOGIC
405
WORD FINDER
410
MULTI-MATCH BUFFER
453
MATCH SORTER
450
470
455
ADDRESS ENCODER
460
OUTPUT BUFFER
CAM_DATA_PIPE
RAM_DATA_PIPE
FIELD_START_PIPE
MASK_SELECT_PIPE
OP_CAM_PIPE
OP-ENABLE_PIPE
READY
RAM_ZERO
WRITE_DISABLE_NEXT
RAM_NEXT
ADDRESS_VALID_NEXT
ADDRESS_MORE_NEXT
ADDRESS_NEXT
DATA OUTPUT
OPCODE OUTPUT
SYSTEM OUTPUT
CASCADE OUTPUT

FIG. 6

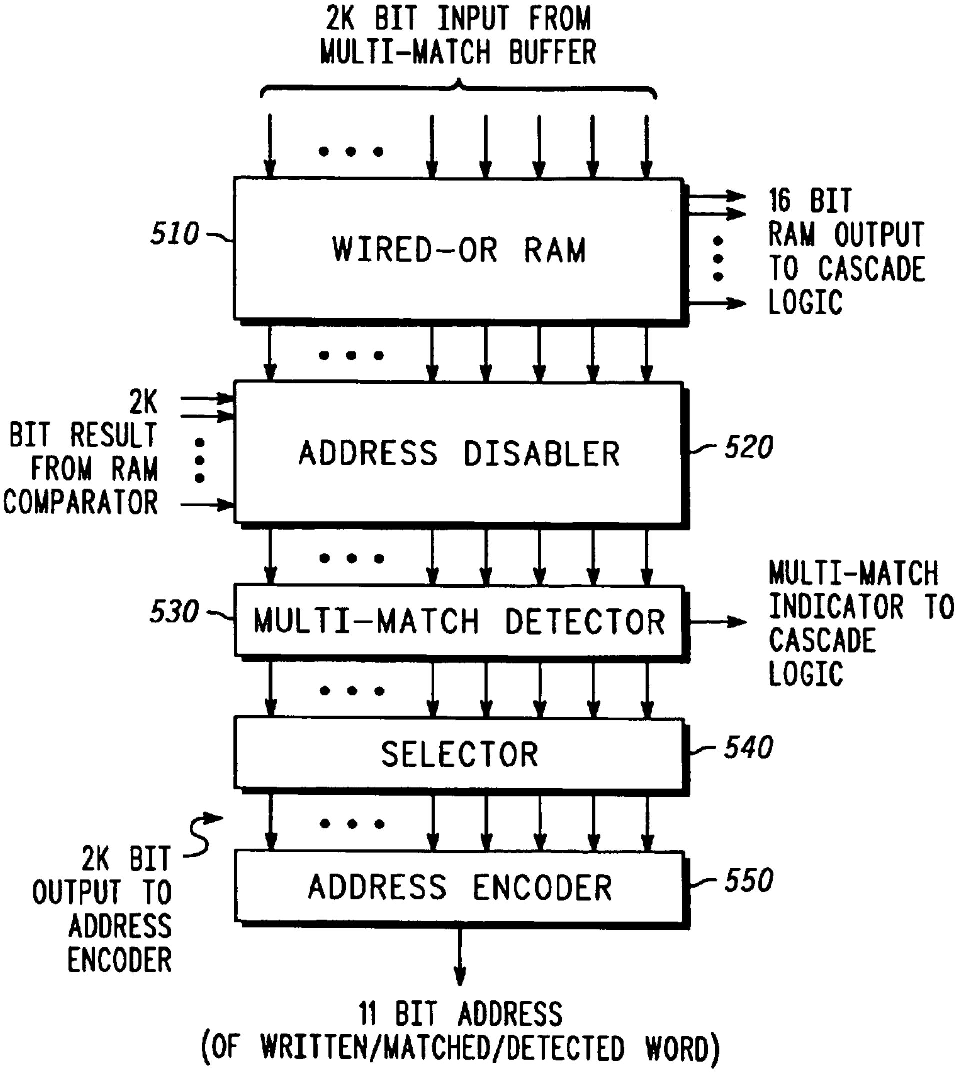

2K BIT INPUT FROM
MULTI-MATCH BUFFER
510
WIRED-OR RAM
16 BIT
RAM OUTPUT
TO CASCADE
LOGIC
2K
BIT RESULT
FROM RAM
COMPARATOR
ADDRESS DISABLER
520
530
MULTI-MATCH DETECTOR
MULTI-MATCH
INDICATOR TO
CASCADE
LOGIC
SELECTOR
540
2K BIT
OUTPUT TO
ADDRESS
ENCODER
ADDRESS ENCODER
550
11 BIT ADDRESS
(OF WRITTEN/MATCHED/DETECTED WORD)

FIG. 7

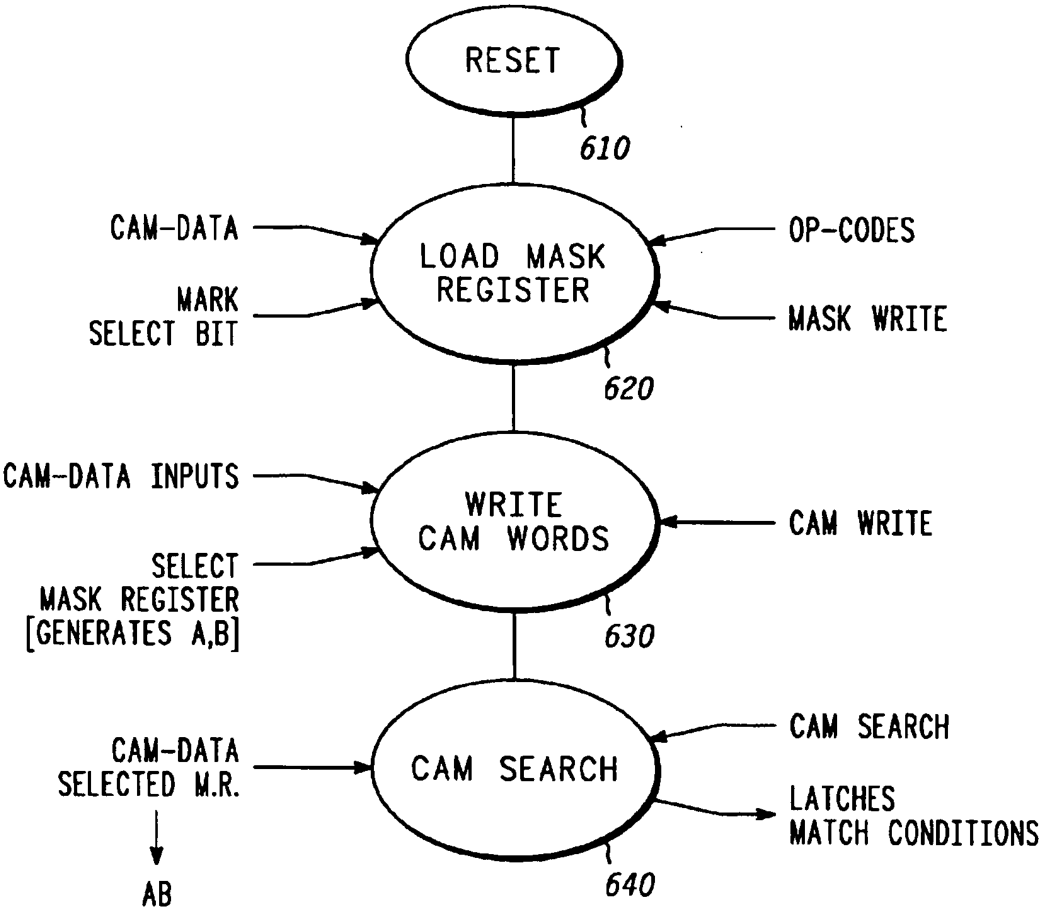

RESET
610
CAM-DATA
MARK
SELECT BIT
LOAD MASK
REGISTER
OP-CODES
MASK WRITE
620
CAM-DATA INPUTS
SELECT
MASK REGISTER
[GENERATES A,B]
WRITE
CAM WORDS
CAM WRITE
630
CAM-DATA
SELECTED M.R.
AB
CAM SEARCH
CAM SEARCH
LATCHES
MATCH CONDITIONS
640

FIG. 8

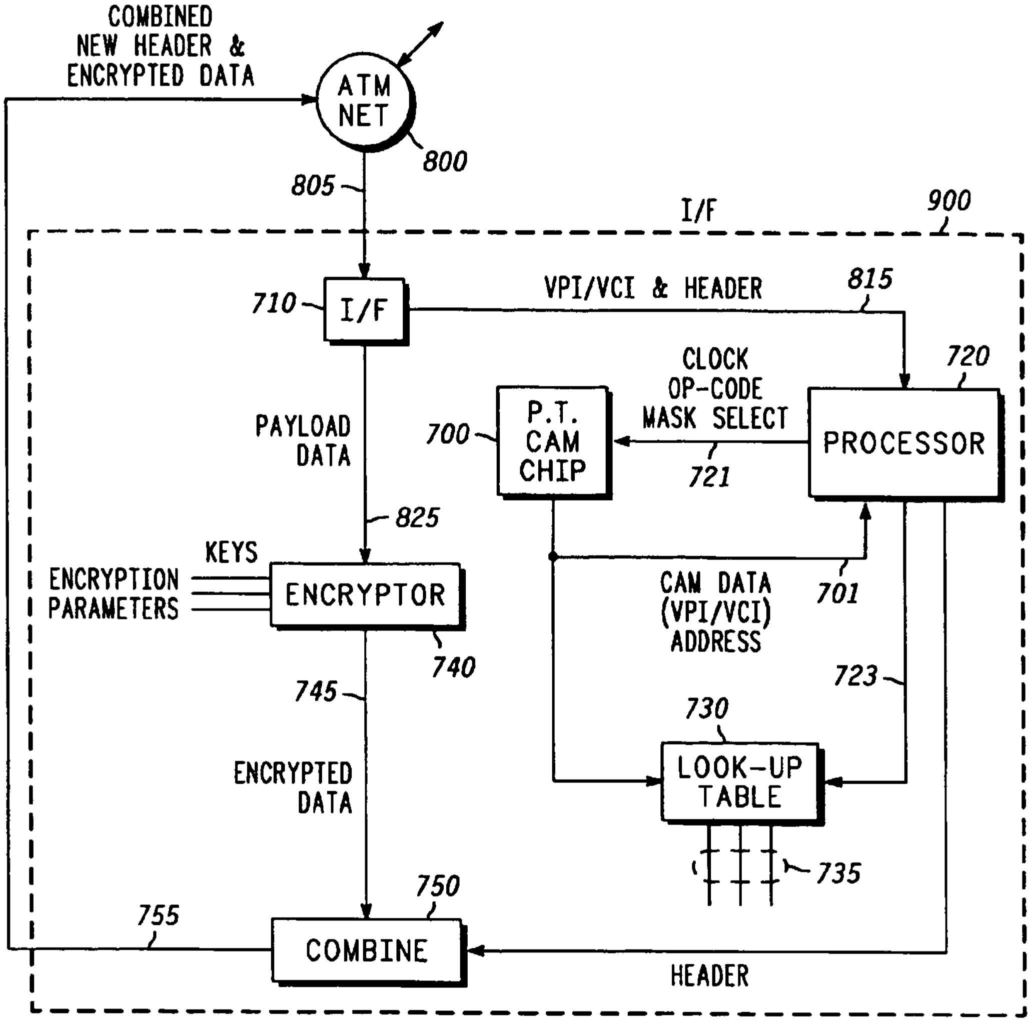

COMBINED
NEW HEADER &
ENCRYPTED DATA
ATM
NET
800
805
I/F
900
710
I/F
VPI/VCI & HEADER
815
720
CLOCK
OP-CODE
MASK SELECT
PAYLOAD
DATA
700
P.T.
CAM
CHIP
721
PROCESSOR
825
KEYS
ENCRYPTION
PARAMETERS
ENCRYPTOR
701
CAM DATA
(VPI/VCI)
ADDRESS
740
745
723
730
ENCRYPTED
DATA
LOOK-UP
TABLE
735
750
755
COMBINE
HEADER

FIG. 9

TERNARY CAM MEMORY ARCHITECTURE AND METHODOLOGY

FIELD OF THE INVENTION

The present invention relates generally to semi-conductor memory, and more particularly, to a cascadable content addressable memory device, and a system using a plurality of such cascaded devices.

BACKGROUND OF THE INVENTION

Content addressable memory devices (CAMs) are extremely valuable in providing associative look-up based on contents of the data. By pre-loading a CAM with a pre-defined data set, including the data to be compared and optionally data to be output when a match is found for a corresponding CAM location, the address where the match is found can be output as an index to the requesting device, or both the address and data can be output for each match. One problem incurred in using CAMs is that the construction of CAM chips requires multiples of the number of transistors needed to implement the CAM memory that a standard read/write random access memory (RAM) would require. Thus, CAM chips are usually much smaller in depth size than RAM chips. Therefore, the capacity of a single CAM chip is frequently inadequate to provide for the necessary associative look-ups. Accordingly, it becomes necessary to use multiple CAM chips in some sort of cascaded or interconnected manner to provide greater depth. Thus, while a single CAM chip, for example, might provide 1k words of 32 bits of CAM capacity, the system's requirements may require 8k or 64k or even 1 megaword of CAM capacity.

Prior system designs have attempted to resolve the depth capacity problem by cascading chips in a seriatim manner where each CAM performed a look up function, and if no match was found, then the next CAM in the cascade chain would attempt its look-up and so forth, looking until a match was found. A major problem with this approach is that there is a variable latency in this architecture, where the time taken to find a match is widely variable from associative look-up to associative look-up, due to the fact that there is uncertainty as to how many CAM chips in the chain will have to be accessed, one at a time in turn, until a match is found. CAM Data Input lines must be run in parallel to all of the chips in the cascade chain, and control logic and intercoupling must be provided between the multiple chips in the cascade chain.

Binary CAM architectures, which store 1 and 0 data, have been the predominant technology used for content addressable memory (CAM) applications because of their speed and density advantages over theoretical ternary CAM architectures, which store 1, 0, and don't care "x" data. This is because each ternary CAM data element contains two memory storage elements, and where static RAM-type memory cells are used, which require data and complement data, two clock cycles are required to write both the data and mask memory elements for each of the ternary CAM cells.

Therefore, a need exists for a ternary CAM memory which can operate in a single clock cycle. With no speed disadvantage over binary CAMs, ternary CAMs could be used for such things as address resolution, filtering, mapping, virtual LANs, Asynchronous Transfer Mode (ATM) communications and systems, etc.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a memory cell in accordance with the present invention for a ternary CAM cell semiconductor;

FIG. 2 illustrates a fourteen transistor static ternary CAM cell, corresponding to the memory cell 100 of FIG. 1, illustrated in further detail;

FIG. 3 illustrates the timing waveforms for the write cycles to the CAM memory, in accordance with the coding provided as illustrated by Table 1;

FIG. 4 illustrates the operation of the memory system during a search cycle, corresponding to FIGS. 1 and 2 and Table 2;

FIG. 5 illustrates a memory array in accordance with the present invention, comprised of multiple memory storage subsystems 100 of FIG. 1;

FIG. 6 illustrates a functional block diagram of a CAM memory system in accordance with the present invention, and illustrates external inputs and outputs to the memory system;

FIG. 7 illustrates a detailed diagram of the match sorter and address encoder of FIG. 6 in further detail;

FIG. 8 illustrates a state flow chart of the operation of the CAM memory system in accordance with the present invention; and

FIG. 9 illustrates a system embodiment of the CAM memory system of the present invention in an ATM data encrypting embodiment.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

In accordance with the present invention, a ternary CAM architecture is provided, comprising a binary-to-ternary conversion subsystem for providing first and second ternary data outputs responsive to binary data input, a memory for simultaneously storing the first and second ternary outputs and providing a content addressable memory, and a comparator for providing a match output responsive to the comparison of binary data currently being input for comparison as compared to the stored first and second ternary data outputs in the memory. In the preferred embodiment, there are a plurality of memory cells, forming a plurality of multiple bit storage words for storing the first and second ternary data outputs for a plurality of words. Furthermore, in the preferred embodiment, a mask register subsystem is provided that is utilized in performing the binary-to-ternary data conversion of data input signals, both for purposes of storing the ternary data outputs in the memory and for purposes of providing ternary data input for comparison to the stored ternary data. In the preferred embodiment, a plurality of addressable mask registers provide for the selection of any one of a plurality of binary-to-ternary conversion options, both for purposes of storage, and independently for purposes of input data conversion for purposes of comparison.

In accordance with one aspect of the present invention, a method of storing ternary data is provided, comprising the steps of (1) initializing a conversion register by storing binary-to-ternary mask data in a conversion register; (2) storing ternary data in a content addressable memory (CAM) by inputting a single bit binary data to the conversion register; and converting the binary data into two bits of ternary data using the conversion register; and (3) simultaneously storing the two bits of ternary data in first and second memory cells.

In one embodiment, for subsequent searching, the method is further comprised of the steps of searching for a match of input search binary data to the stored contents of the CAM; providing a match valid output responsive to the input search binary bits matching any of the stored contents; and generating an address ~~corresponding to a~~ location in the CAM where the match is found. The step of searching for a match is further comprised of the steps of converting the input search binary data into ternary search data, using the conversion register; and comparing the contents of the CAM to the ternary search data.

Referring to FIG. 1, there is illustrated a memory cell for a ternary CAM cell 100 in CAM memory subsystem 160, in accordance with the present invention. An externally provided Opcode input is decoded by decoder 110 to provide control signals for Mask Write, CAM Write, and other control signals to other areas of the chip, as will be described in greater detail hereafter. The CAM cell 100 is comprised of a first memory cell 120 and a second memory cell 130, which are utilized to store the ternary data for that CAM cell location. Memory cell 120 stores the 1 or 0 data values, while memory cell 130 stores a 1 for a don't care "x" value selection, and a 0 to indicate that the memory cell contains data that should be utilized. An external clock is provided to the subsystem 160, where the clock signal 105 is utilized to synchronize operations within the CAM memory. Additionally, external Mask Select code signals and Data Input signals are provided to the CAM memory 160 and are coupled to a binary-to-ternary converter 140 that converts the binary Data Input into appropriate ternary data signal outputs A and B, which respectively couple to memory cells 120,130. The Mask Write, CAM Write, and CAM Search outputs from the decode logic 110 are coupled to the binary-to-ternary converter 140. These outputs from the decode logic 110 selectively control (1) the writing of the Data Input into the mask registers and into the binary-to-ternary converter, responsive to the Mask Select code; (2) the conversion of the Data Input by the binary-to-ternary converter into ternary code; (3) the writing of the ternary code into the memory cells 120 and 130 responsive to the CAM Write decode output; and (4) the activation of a CAM search for comparing the ternary code for the then present current Data Input to the CAM cell contents by a comparator 150, which is clocked at the appropriate time to do the comparison.

The Mask Select code, Data Input, Clock, and Opcode inputs are provided to the CAM memory 160 from an external device or subsystem, such as a processor, which controls and establishes the loading of the mask registers in the binary-to-ternary converter 140, the loading of the CAM data contents into the memory cells 120,130 and the setup and request for comparison. The comparator 150 provides a Match output indicating that the Data Input either matches or doesn't match the contents of the CAM memory. The binary-to-ternary conversion subsystem 140 provides the first and second ternary data outputs (A and B) responsive to the Mask Select signals, the Data Input signals, and the Opcode signals. The memory cell 120 stores and selectively outputs the first ternary data output A, responsive to the binary-to-ternary conversion subsystem 140 and to the Opcode signals. The memory cell 130 stores the second ternary data output B, responsive to the binary-to-ternary conversion subsystem 140 and the Opcode signals. The comparator 150 performs an exclusive-OR comparison in the preferred embodiment, and provides a match output responsive to comparing the first and second ternary outputs provided by the binary-to-ternary converter 140 for the then current Data Inputs, to the outputs of the first and second memory cells. Since the data storage to memory cells 120,130 is performed in parallel, that is, the binary-to-ternary converter provides output A to memory cell 120, and output B to memory cell 130, the memory (CAM memory subsystem) device 160 is able to store the first and second ternary outputs (A and B) in the CAM cell 100, in parallel, within a single period of the clock's signal. This advantageously provides for single cycle ternary data conversion, comparison, and write cycles. The ternary CAM storage and retrieval system ("TCAM") is inherently superior to binary CAMs in masking operations, and useful in storage and/or conversion, and/or comparison. The TCAM system is useful in a multitude of applications, such as filtering, address resolution, mapping, Virtual LANS, etc.

Referring to FIG. 2, a fourteen transistor static ternary CAM cell, corresponding to the CAM cell 100 of FIG. 1, is illustrated in further detail in accordance with one embodiment of the present invention. The first and second ternary data outputs (A and B) from the binary-to-ternary converter 140, are coupled to the CAM cell 100. The memory cell 120 receives the A input and, responsive to the CAM Write signal, is output from the decode logic 110, enables transistor 210 to couple the A signal to the storage element portion of the memory cell 120, as appropriately timed and clocked responsive to the external Clock, as discussed in further detail with reference to FIG. 3 for the timing diagrams for the write cycle. Transistors 220–223 form the static memory cell storage element 120. The output RA from the memory cell 120, representing an inversion of the originally input signal A, is coupled to one input of the comparator 150, as is described in greater detail hereinafter.

Similarly, the second ternary data output B is coupled to the memory cell 130 of the CAM cell 100 and is coupled via transistor 230, responsive to the CAM Write signal output from the decode logic, via transistor 230, to the memory storage cell 130 comprised of transistors 230–233, forming the storage element portion of the memory cell 130. The storage cell output from memory cell 130, output RB, provides an output corresponding to the inverted value of B as stored initially to the memory cell 130. The output RB is coupled to a separate input of the comparator 150.

In a preferred embodiment, the comparator 150 is comprised of a four transistor exclusive-OR comparator. The comparator 150 is comprised of two subportions, subportion one comprised of transistors 251 and 252, and a second subportion comprised of transistors 253 and 254. On a CAM search cycle, the then current search Data Input is converted by the binary-to-ternary converter to provide the first and second ternary data outputs A and B, which are coupled to the comparator 150 transistor gates for transistors 251 and 254, respectively. Correspondingly, the CAM memory cell outputs RA and RB are coupled to the gates of transistors 252 and 253, respectively. Thus, the ternary data for a search is coupled to the gates of one transistor of each portion of the comparator and the corresponding stored ternary data output from the CAM cell 100 is coupled to the gates of the other transistor of the corresponding portions of the comparator. Each portion forms a transistor series tree, having one side coupled to ground (e.g., the source of transistors 251 and 254) and having the other side of the tree coupled to a Match line output 170. The transistors on each side of the series tree are series connected, that is the other side (the drain) of transistor 251 is coupled to the source of transistor 252, and the drain of transistor 252 is coupled to the Match line 170 and to the drain of transistor 253, which has its source coupled to the drain of transistor 254, which has its source coupled to ground. The Match line is pre-charged during the first portion of each clock cycle to a high level, and thereafter is pulled to ground, or low level, if no match is found, thus indicating no match. If there is a match, none of the transistors in the comparator **150** are enabled and the match line stays at a high level.

It is important to note that the ternary converter outputs, A and B signals, are not bit and bit bar, but are rather the encoded two-bit state to be written into the two memory element CAM cells **120**,**130**. Thus, the standard six transistor static RAM cell, which writes bit and bit bar into both sides of the memory element, is not utilized, but rather a five transistor memory cell is utilized that provides a single ended input structure, which is driven by the A line for memory cell **120** or the B line for memory cell **130**.

The comparison circuit **150**, as illustrated, is comprised of four N-channel transistors. On one side (one portion) of the comparator, the gates of each of the two transistors are tied to the A line and RA lines (transistors **251** and **252**). On the other side (the other portion) of the comparator, the gates are tied to the B line and the RB line (transistors **254** and **253**, respectively). On one side of the comparator, a comparison is made, between the incoming decoded A signal and the memory cell output RA signal (by transistors **251** and **252**) and on the other side of the comparator, a comparison is made between the B signal and the RB signal (transistors **254** and **253**, respectively). The binary-to-ternary converter **140** performs a code conversion as provided in Table 1 for a writing table (where the Write line equals one), and as provided in Table 2 for a matching table, where the Write line equals 0. For either a write or search operation, the binary Data Input is encoded by the binary-to-ternary converter **140** to corresponding A and B outputs. The Tables 1 and 2 show four ternary codes for conversion. In a preferred embodiment, the null state "N" is not used for writing or searching, and is used for pre-charge and test functions only.

Table 1 illustrates the writing table where the write line equals one, providing the logic for the binary to the ternary converter **140** performing code conversion;

TABLE 1

(B -> T Conversion Table)

Ternary	A	B	RA	RB
"N"	0	0	1	1
"1"	0	1	1	0
"0"	1	0	0	1
"X"	1	1	0	0

As shown, Table 1 depicts the ternary symbol (N,1,0,X) and the corresponding ternary data outputs A and B, and the corresponding memory cell outputs RA and RB.

Table 2 provides the code conversion for the matching operation for the binary-to-ternary converter **140** performing code conversion;

TABLE 2

Matching Table (Write = 0)

Ternary	A	B	RA	RB	MA
"X"	0	0	X	X	1
"1"	0	1	X	0	1
				1	0
"0"	1	0	0	X	1
			1		0
"N"	1	1	0	0	1
			ELSE		0

As shown, Table 2 illustrates the matching table, showing the ternary symbol (N,0,1,X) and the corresponding ternary data outputs A and B, plus showing the Match output **170** resulting from a comparison of the ternary code for the input data to the stored memory cell output data (RA and RB).

By way of example, and referring to Table 2, in conjunction with reference to the comparator **150** of FIG. **2**, the search data A is compared to the stored cell output RA, and if they are exclusive of each other, that is, if only one of them is logic high, this will inhibit that side of the comparator (transistors **251** and **252**) from pulling the match line low. Similarly, if RB and B (the search data B) are compared, and only one of them is high, indicating that there is a match, then that side of the transistor series tree will not be able to pull the match line low. When neither one of the sides of the transistor series trees **251** and **252**, or **254** and **253**, are able to pull the match line low, this indicates that there is a match in the cell. Conversely, if there is not a match in the cell, for example, if both A and RA are high, this would turn on both transistors **251** and **252**, which would connect the output point coupling **252** and **253** to ground, thus pulling the match line low, thus indicating there is not a match. Where there are a number of memory cells corresponding to a single memory word, the match line for each memory cell is coupled to other match lines for that single word in a wired-OR structure, such that a number of memory elements can be cascaded together to form a word, and if any of the memory cells forming the multiple bit word don't match, it will pull the match line low, indicating no match.

Referring to FIG. **3**, the timing waveforms for write cycles to the CAM memory is illustrated for a write zero cycle and a write one cycle. First, it is noted that the coding of the A and B lines, as indicated in Tables 1 and 2, are such that only one of the A or B lines will transition at a time during write and search operation.

As illustrated in FIG. **3**, there is a Clock (i.e., the Clock signal **105** of FIG. **1**) which periodically cycles at a system clock rate. During the first portion of the clock cycle, a pre-charge (or setup) cycle occurs (i.e., a pre-charge signal goes high) to pre-charge the match line, and both the A and B lines are forced to an active low state, putting 0 on the A and B lines during pre-charge. This guarantees that independent of whatever data is stored in the memory cells **120** and **130**, that the match line can be pre-charged to a logic one without having any DC path in the exclusive-OR circuitry of the comparator **150**. After the pre-charge cycle is over (i.e., the pre-charge signal goes low), the A and B lines are allowed to change. The match line must be pre-charged, since if the match line is pulled low, it will remain low, since there is no active device to pull it up. Thus, the match line has to be reestablished at a high level each cycle, to establish a base level of one for a match each time. After the pre-charge signal goes low, a search operation can begin. This will be discussed in further detail with reference to FIG. **4**. The pre-charge is not directly used in the write cycle, and is used directly for the search cycle, as illustrated in FIG. **4**. However, the pre-charge period is relevant in that the A and B lines are forced low during the pre-charge cycle time.

As illustrated in FIG. **3**, after the pre-charge cycle (the pre-charge clock signal goes from 1 to 0), a write operation may be performed. As illustrated in FIG. **3**, a write "0" cycle occurs after the first pre-charge cycle during the first Clock cycle. The A and B signals are set up by the binary-to-ternary converter **140** and coupled to the memory cells **120** and **130** as illustrated in FIG. **1**. Thus, as set forth in Table 1, to write a 0, the ternary code outputs A=1 and B=0, are provided, as illustrated in FIG. **3**. The RA and RB signal lines are initially at a 0 level, and upon the Write CAM signal (**115** of FIG. **2**) being clocked high, the 0 write data is clocked into the memory cells **120** and **130** and correspondingly, output RA=0 and output RB =1, as illustrated in FIGS. **2** and **3**, and Table 1. Referring to the Clock signal of FIG. **3**, the Clock then transitions from 1 to 0, again initiating the pre-charge pulse to initiate the pre-charge cycle, forcing the A and B lines to a 0 level, and thereafter, when the pre-charge pulse goes back to a 0 level, the A and B ternary code outputs are set up for writing a 1. As shown in FIG. **3**, consistent with the writing table of Table 1, the ternary code for a "1" is A=0 and B=1. Upon the Write clock **115** of FIG. **2** clocking active high, the ternary code is written into the memory cells **120** and **130**, respectively, storing inputs A and B, and the memory cell outputs provide for an output of RA=1 and RB=0, consistent with Table 1, illustrating the operation of the CAM cell **100** of FIG. **2**. The writing of a "don't care" would be similar to that illustrated for the writing of a 0 and 1, except that in this case, both the A and B signals would be at a high level (i.e., A=B=1), which would be clocked upon the Write clock signal going high into the memory cells **120** and **130**, correspondingly thereafter providing outputs RA=RB=0. Similarly, writing an "N" into the memory cell would write a 0/0 (A=B=0), providing a corresponding 1/1 output from RA/RB (i.e., RA=RB=1), leaving it to the incoming A and B to determine if there's a match, since transistors **252** and **253** of comparator **150** would already be enabled (by RA=RB=1) for a comparison.

Referring to FIG. **4**, a search cycle is illustrated in accordance with a preferred embodiment of the present invention. This assumes that the memory cells **120** and **130** have already been written to during a previous write operation. FIG. **4** illustrates only the search operation. The external input of Data Inputs and Mask Select inputs, in conjunction with the Opcode input, cause the binary-to-ternary converter **140** to provide ternary code data A and B outputs, which are ultimately fed to the comparator **150**, appropriately timed as illustrated in FIG. **4**. Analogous to FIG. **3**, FIG. **4** illustrates the periodic Clock signal, Pre-charge clock signal, the ternary code outputs A and B, and the memory cell outputs RA and RB, and also illustrates the Match line output **170**, corresponding to FIGS. **1** and **2** and Table 2. The timing diagram of FIG. **4** illustrates first a search for a 1 and then a search for a 0. During the first portion of the Clock cycle, illustrated as the first quarter of the Clock cycle, the A and B lines are forced low, and the pre-charge line is charged to a high level (which high level indicates a Match if it still exists after a search is done). At the end of the pre-charge clock, the A and B signal lines are set up with the appropriate data, and since this is a search for 1, A=0 and B=1, in accordance with Table 2. This is compared to the stored memory cell outputs RA and RB by the comparator **150** as illustrated and described with reference to FIG. **2**, and at the end of the first clock cycle, the Match line status is clocked into a buffer, and the buffer output is then decoded to determine the address where the match occurred (where there are multiple memory words in the CAM).

When the Clock goes low to start the next Clock cycle, the pre-charge signal goes high, forcing the A and B ternary code outputs low, to permit the Match line to pre-charge back to 1 if needed. Then, when the pre-charge clock signal goes low, a search for 0 is set up, with A=1 and B=0, as set forth in Table 2. Since the stored memory cell output RA=1 and RB=0, then both A=1 and RA=1, thus causing transistors **251** and **252** of FIG. **2** to couple the Match line to ground, thus forcing the Match signal low as illustrated in FIG. **4**, indicating no match has occurred. As illustrated in FIG. **4**, a buffer clock signal occurs at the end of each of the Clock cycles, clocking the match status into a Match Buffer.

The Buffer stores the match state for each Match line word location in the CAM memory.

Also, note that if RA and RB are low, regardless of what A and B are on subsequent search cycles, there will always be a Match output. Additionally, if the external system Data Input is converted into a ternary code of 0/0 for A and B (i.e., A=B=0) for a search, indicating a don't care, then the comparator will provide a Match output no matter what is stored in the memory cells. This permits a global don't care search. Thus, two don't care search options are provided. Initially, the system can process the data coming in and store don't cares into the memory. Subsequent to that, the system can also mask out certain bits of information in the CAM, using a don't care ternary code input for the search data. Thus, a "don't care" search data input ignores what's stored in the CAM cell, and causes the respective don't care bit locations to always match. It is to be understood that there are multiple bits in the word (64 bits in the preferred embodiment), and thus individual bits can be masked for don't cares during search in real time, or can be pre-stored as don't cares when loading and writing the CAM memory cells, so that a match is found irrespective of the search data for that bit.

In a preferred embodiment, as illustrated in FIG. 5, the CAM memory **300** is comprised of a plurality of CAM cells **100** cascaded both in rows as words (of 64 bits each), and as multiple rows comprising multiple (64 bit) words. Each word is comprised of **64** CAM cells **100**. Thus, where 64 memory subsystems are coupled together to create a 64 bit word, all 64 of these match lines are coupled together in a wired-OR structure, so that any element on the match line has the capability of pulling the match line low. Thus, all 64 bits would have to match in order for the final Match output for that word to be valid (high).

Since the comparator **150** is an exclusive-OR type device, the inverted outputs RA and RB are compared to the non-inverted ternary code outputs A and B so that inverted stored ternary code data is compared by the comparator to the non-inverted ternary code search data. The effect of the exclusive-OR is to utilize the inverted to non-inverted comparison to affect a comparison that utilizes an inverted and non-inverted input to properly perform the comparison. Other comparator structures can alternatively be utilized, which use either inverted outputs or non-inverted outputs from the memory cells **120** and **130**, as inputs to the comparator, and the choice is one of design alternatives.

Referring again to FIG. **5**, a memory array comprised of multiple memory storage sub-systems **100** is illustrated, wherein there are a plurality of memory subsystems **100** coupled in a row forming a word, illustrated as 4 bits wide (64 bits wide in the preferred embodiment) having a common wired-OR match output (match 0) which is coupled to a multi-Match Buffer register. There are a plurality of rows of memory subsystems **100**, forming a plurality of words each n bits wide, each row coupling its match line in common to all memory subsystems **100** of that row (memory word) providing corresponding match outputs (Match (1) to . . . Match (N)), which Match outputs are coupled to separate flip-flops (or other storage) of the multi-Match Buffer register **310**. The array of memory subsystems **100** form the memory array **300**, in the preferred embodiment being 64 bits wide by 2048 words deep. The multi-Match Buffer register **310** provides a valid Match output **370**, which indicates a valid match has occurred, and one output for each CAM location, indicating that location's match status. Thus, the depth problem is solved, and both multiple bit words and multiple word arrays of ternary CAM storage are efficiently provided without speed operational penalties.

The CAM memory system of FIG. 5 includes the memory array **300**, the multi-match buffer register **310**, and the address generator (encoder) **320**, plus other logic as described elsewhere herein. The multi-Match Buffer register provides a plurality of outputs, corresponding to the match status for each row or word of the memory array **300**, and the Buffer register **310** outputs are coupled to the address generator **320**, which performs a decimal to binary encoder function to provide an address output **325**, representative of the address where the first match occurred. In the preferred embodiment, the encoding is from 2048 registers to a 10 bit binary code. This then provides an address output from the CAM memory system which indicates the address corresponding to the first valid match.

Referring to FIG. **6**, a functional block diagram of a CAM memory system is illustrated, wherein external system inputs are provided as Data Input, Opcode Input, System Input, and Cascade Input, and where the memory system output is comprised of a Data Output, Opcode Output, System Output, and Cascade Output. Self testing is possible through the use of the Opcode Input and System Inputs, and can be performed by a self-test sub-system **405**, and can be optionally provided depending on design considerations. The data coming in through the Data Input is coupled through the input buffer **410** to mask registers **440**, which provide for binary-to-ternary conversion, in accordance with Tables 1 and 2. A host processor system provides the Opcode and Data Inputs necessary to load the mask registers to determine when don't cares are to occur. Thus, initially the host processor provides appropriate Mask Select inputs and CAM Data Inputs to load the mask registers. In a preferred embodiment, 8 mask registers are used, and a 3 bit Mask Select input can provide selection of one of the 8 mask registers. Upon completion of loading of the mask registers, the host system provides either a 1 or 0 (binary data), and a Mask Select code, and the binary-to-ternary converter converts the binary input into ternary code data as described earlier with reference to Tables 1 and 2 and FIGS. 1 and 2. The mask register is the width of the CAM words, 64 bits in the preferred embodiment, and any one of those bits, or any combination of those bits within each mask register, can be set to a logic 1 to mask out any bit location within the word being written into the CAM as a don't care. If there is a 0 in the mask register, then the data (i.e., 0 or 1) is appropriately written into memory cell 120 of FIG. 1. When selected during CAM Write, the CAM memory is written into so that memory cell **120** is written with a binary 1 or 0, and dependent on the mask register bit, a 1 or 0 is written into memory cell **130**, indicating either a don't care where a 1 is written into memory cell **130** for those mask register bits having a 1, or that valid data (i.e., a 1 or 0) exists in memory cell **120** where a 0 is written into memory cell **130**. In one embodiment, RAM storage is provided as a part of the CAM memory system, illustrated as RAM **450**, which provides storage corresponding to each word of the CAM array. In the preferred embodiment, 2K words by 16 bits of RAM are provided corresponding to the 2k words of CAM. However, the width is determined by design constraints, in a preferred embodiment, 16 bits.

The RAM is loaded by the host system (or other external device), using the Opcode Inputs, and the RAM Data Inputs of the Data Input, to provide for loading of the RAM with contents that are thereafter provided as outputs for matched CAM locations, responsive to the multi-Match Buffer **410** and the RAM comparator **420**, to provide RAM data output (for the matched Cam location) to the output buffer **460**. The output buffer **460** provides the Cascade Output for a multiple memory system chip cascaded configuration including the RAM next output, the address output indicative of a match being found, and other control signals to permit cascading, as well as feed-through of the Data Output corresponding to the Data Input via CAM data pipe **0–63**, and RAM data pipe **0–15**. Additionally, the output buffer **460** provides feed-through of the Opcode input including Field Start pipe, Mask Select pipe, OpCAM pipe **0–3**, and OpEnable pipe **0–2**. The System Output feeds through the Clock, and the Ready, and RAM Zero System Output. The outputs provide for multiple ones of the memory chip systems **500** for coupling to one another to form an array (e.g., having a depth greater than 2K), but providing for output of the address and data (where the RAM is present within the system **500**) to provide the indication of the address and corresponding RAM data for the first match location. The cascade logic **470** provides for cascade logic and feed-through of appropriate signals from the Cascade Input, the RAM, the multi-Match Buffer, and the address encoder to provide for the Cascade Output. The address encoder **455** is coupled through the Match Sorter **453** to the multi-Match Buffer **410**, to provide a decode of the Match Buffer **410** and provide the corresponding address where the match was found. The Word Finder logic **405** provides necessary decode logic between the CAM memory array **300** and the multi-Match Buffer **410** to provide appropriate logic for interfacing to the multi-Match Buffer **410** and the RAM comparator **420**.

Referring to FIG. **7**, the main components of the match sorter and address encoder of FIG. **6** are provided in greater detail. The 2 kilobits of output from the multi-match buffer **410** are coupled as inputs to a wired-OR RAM **510**, which provides a 16 bit RAM output to the cascade logic **470** as shown in FIG. **6**. The wired-OR RAM **510** corresponds to the RAM array **450** and RAM comparator logic **420** of FIG. **6**. The output of the wired-OR RAM is also coupled to an address disabler **520**, which selectively disables the address output responsive to the output from the RAM comparator **420**. The address disabler output selectively provides an address disabling output to the multi-match detector **530**, which provides an indication of a multiple match indicator to the cascade logic **470**, and detects if more than one of the 2K lines are active. The selector **540** picks the first matched word and disables all others (the lowest address having the highest priority in the illustrated embodiment), and provides 2 kilobits of output to the address encoder to be encoded into an 11 bit address output from the address encoder **550** (as discussed in detail with reference to FIG. **5** address generator encoder **320** and analogous to the address encoder **455** of FIG. **6**) indicative of the written, matched, or deleted work, depending on the type of operation being provided.

Referring to FIG. **8**, a state flow diagram is provided illustrating the operation of the system in resetting, loading mask registers, writing CAM words, and performing a CAM search. Initially, at state **610**, the external system provides for a chip Reset that initializes everything (e.g., all memory cells and mask registers) to 0. Thereafter, the next state **620** provides for loading of the mask registers. In the preferred embodiment as discussed above, there are 8 mask registers, however a single mask register can be used multiple times, and the choice is one of design alternatives. Responsive to the Opcode Inputs and Mask Select bits, and the Mask Write signal, the CAM Data Input is written into the mask registers at step **620**, as indicated via the signal inputs coupling into step **620** from the left and right. To avoid applying any mask to the CAM Data being written in, one mask can be reserved and loaded with all 0's, which provides for no masking function. When writing to the CAM, at step **630**, the Opcode combination provides appropriate signals, and the Mask Write is inactive, indicating a Mask Write is not occurring, and that the write cycle is a CAM data write. The CAM Write signal is active, and the CAM Data Inputs are coupled into and written via a selected mask register as determined by the Mask Select bits. This results in the binary-to-ternary converter generating the appropriate A/B signals that go into the CAM memory elements for storage. Once the CAM memory array has been written to and the data stored (and where a RAM is located internal to the CAM memory system the RAM is also loaded as appropriate via RAM Data Inputs and appropriate Opcode signals), a CAM Search operation can be provided. The CAM search operation is responsive to a CAM Search Opcode, and the CAM Data Input, and a selected mask register as selected via Mask Select bits, which, as shown at step **640**, provides for determining whether there is a match between the then current input CAM Data as compared to any stored value in the CAM memory array. If a match is found, the corresponding match lines are latched (in the multi-Match Buffer) and the output from the chip system is an address indicating where the match occurred. Additionally, where RAM is present on the chip, the RAM data for the corresponding CAM word location where a match was found is also output from the chip.

Referring to FIG. **9**, the memory system of the present invention is illustrated in an encryption embodiment for use in conjunction with an Asynchronous Transfer Mode (ATM) system. During an initial call setup, the ATM network **800** provides for communication of information coupled via bus **805** to interface **710** to establish a call setup procedure prior to performing a write operation. The ATM memory system embodiment system **900** provides for storing of new ATM virtual address (i.e., as commonly denoted by its Virtual Pipe Identifier (VPI) and Virtual Channel Identifier (VCI)) link data to be setup and stored into the CAM memory array of memory system **700** by performing the CAM Write cycle process. In accordance with the present invention, a ternary CAM system is provided that provides for ternary information being written into the ternary CAM cells in a single Clock cycle, which allows for the writing of a continuous stream of ATM messages, instead of having to stall or delay the ATM system to facilitate a multiple cycle ternary CAM Write with risk of cell loss. In typical applications, an entire block of VPI/VCI link translation address information is setup in the CAM memory cells, the lookup table, and the internal RAM if present, all in one continuous set of operations rather than just one location. A real-time communication network is thereby provided.

motivation

After initial setup, communications from the ATM net **800** via coupling **805** is made to an interface **710**, which strips off the VPI/VCI portion of the header from the payload and remaining header potion of the ATM cell, and sends the VPI/VCI and remaining header, via coupling **815**, to the processor **720**. The processor **720** provides the appropriate Clock, Opcode, Mask Selects, CAM data, and other appropriate input signals via coupling **721** to the ternary CAM memory system **700**. The ternary CAM memory system **700** can be comprised of one or a plurality of cascaded CAM memory systems of the type discussed elsewhere herein. After setup is complete, the CAM search (and lookup table) can be utilized.

The CAM Data from the processor, which is requesting a compare, is the stripped-off VPI/VCI portion of the header, which is compared to the contents of the CAM memory **700**, which in turn provides an address output **701** when a match occurs. The address output **701** is coupled back to the processor **720** and to a lookup table **730**. During setup, the processor **720** loads the lookup table **730** with data, via coupling **723**, corresponding to the Address output of the CAM **700**. The lookup table **730** outputs specific encryption parameters **735** responsive to the address output of the CAM memory system **700**. The lookup table **730** provides the encryption parameters **735**, which can be a unique key or some mechanism that sets up an encryptor **740**. The encryption parameters **735** are coupled to the encryptor **740**, which is also coupled to receive the payload data portion of the cell **825**, as provided by the interface **710**. The encryptor **740** then encrypts the payload data in accordance with the specific encryption parameter keys as provided by the lookup table **730**, which are uniquely associated with the specific VPI/VCI address that was input as CAM Data into the CAM system **700**. The encrypted data output **745** from the encryptor is coupled to a combiner **750**, which recombines the encrypted data of the payload with the header, including the VPI/VCI address, and provides a combined new cell comprising the header and encrypted data as output at **755** for coupling back to the ATM network **800** for communication therefrom to the appropriate destination.

The lookup table **730**, while illustrated external to the CAM memory system **700**, can alternatively be provided as a part of the CAM memory system **700**. However, to provide sufficient encryption parameters, it is desirable to have more than a 16-bit wide amount of RAM. Thus, to maintain cost effectiveness of the CAM memory chips of the memory system **700**, the lookup table can be provided externally and addressed responsive to the address output from the CAM memory system **700**, to add flexibility to the system design. The RAM within the CAM chip itself, where present, can be used to provide sync pulses, end-of-frame indicators, and many other simpler functions than the encryption parameters, and can be provided in addition to the lookup table **730**. Thus, the presence of the RAM within the CAM memory system **700** is optional, and if present, can be supplemented by an external separate lookup table. Since not every CAM address needs to have a lookup table encryption, an external lookup table can be used with a much denser lookup function than an on-chip RAM. In one embodiment, the RAM is on-chip within the CAM memory system **700**, and the lookup table is integrated internally, eliminating the need for the external lookup table **730**.

The lookup table is loaded as appropriate, corresponding to the CAM cell loading, via the processor **720**, monitoring when a write operation is performed into the CAM memory **700**, and then providing a CAM address output **701** from the CAM **700**, which indicates the memory location that is actually written to. Subsequent to that, the processor **720** takes the appropriate action to load in the lookup table an appropriate mapping of the encryption parameters as necessary to support that VPI/VCI address. Even where the lookup table is in RAM internal to the CAM memory system **700**, the processor still monitors and rewrites into the RAM appropriately to load the encryption parameter data needed. The processor **720** provides the Mask Select, Data Input, the Opcode Data input, the Clock, and other necessary parameters for use by the CAM memory system **700**. The processor **720** processes the VPI/VCI and remainder of the header, and determines the next appropriate step. In the preferred embodiment, the VPI and VCI portion and the remainder of the header are typically not encrypted or transformed by the encryption system as illustrated in FIG. **9**, and are recombined with the encrypted data by the combiner **750**. Alternatively, the VPI/VCI could be re-mapped via the processor and VPI/VCI mapping contained either within the CAM system **700** as RAM or utilizing another external memory system, to provide a new VPI/VCI address to be recombined with the remaining original header and the encrypted data.

The encryptor **740** provides a method of scrambling the input data based on certain encryption parameters, which can be any sort of scrambling and encryption, such as keys for a specific user path. The encryption parameters in the lookup table are thus loaded in accordance with some predefined encryption algorithms to provide the necessary parameters for the encryptors **740**. The keys are loaded as appropriate, so that each respective VPI/VCI address has associated with it its own key, or no key, so that the corresponding destination address system can decode the encrypted data on the other end with that unique key. The lookup table must provide the appropriate equivalent key, so the encryptor encodes the payload data in accordance with the key that is going to be used on the other side when the payload data is decoded.

During the initial call setup from the ATM network, messages are passed back and forth to define what keys (e.g., encryption parameters to be stored in the lookup table) can be used, what algorithms, which VPI/VCI locations have access, and various other parameters that can be defined for the encryption process. An agreed-to initial key can be used to encrypt the initial data that is sent with a common public key that all users have, and thereafter, private keys are utilized for encryption and decoding. The private key is unique for a VPI/VCI pair, although multiple VPI/VCI pairs can have the same key. The processor **720**, responsive to the loading of the CAM, provides for loading the lookup table with the corresponding keys for certain addresses in response to communications from the ATM network **800** of key values for certain VPI/VCI addresses. The interface **710**, the ternary CAM memory system **700**, and the processor **720** provide translation of the VPI/VCI addresses to addresses for encryption keys for the respective VPI/VCI addresses, responsive to the ternary CAM **700** output **701**. The output **701** provides the address to the lookup table **730** which provides the encryption parameters **735** as necessary to encrypt the payload data **825** by the encryptor **740**. The encryption payload data is combined by the combiner **750** with the header for output **755** to the ATM network **800**.

The ATM system benefits by utilizing off-loaded key encryption of payloads, independent of address routing information (e.g., VPI/VCI), which is first stripped, and, after encryption, re-appended from/to the payload. This encryption of payloads can be performed transparently to the ATMs' other network operations. This benefit can also be utilized by other communications schemes, where a portion is stripped off, is encrypted, and then recombined for transmission. On the receiving side, the same associative lookup/mapping is used to determine the encryption keys, and the encrypted payload is then de-encrypted using the encryption keys.

What is claimed is:

1. A memory apparatus for coupling to an external device, the external device outputting (a) a plurality of data input signals to be processed in the memory apparatus, the plurality of data input signals comprising at least a first data input signal and a second data input signal, (b) mask select signals for specifying a data mask indicating desired locations of don't care bits for the data input signals, and (c) opcode signals for specifying operations to be performed by the memory apparatus, the memory apparatus comprising:

a binary-to-ternary conversion subsystem that generates first and second ternary data outputs for each of the plurality of data input signals based upon the mask select signals, the plurality of data input signals, and the opcode signals; and

at least one memory subsystem, comprising:

a first memory cell for storing the first ternary data output for the first data input signal in response to the binary-to-ternary conversion subsystem and the opcode signals, said first memory cell having an output;

a second memory cell for storing the second ternary data output for the first data input signal in response to the binary-to-ternary conversion subsystem and the opcode signals, said second memory cell having an output; and

a comparator for comparing, in response to the opcode signals, the first and second ternary data outputs for the second data input signal to the outputs of the first and second memory cells, said comparator outputting a first match output.

2. The memory apparatus as in claim **1**, further comprising:

means for supplying a periodic clock signal;

wherein the memory apparatus stores the first and second ternary data outputs for the first data signal within one period of the periodic clock signal.

3. The memory apparatus as in claim **1**, wherein the binary-to-ternary conversion subsystem is further comprised of:

a mask register subsystem for selectively storing the data input signals during a setup cycle and for thereafter encoding the data input signals into the first and second ternary data outputs based on the mask select signals.

4. The memory apparatus as in claim **3**, wherein the mask register subsystem is comprised of a plurality of mask registers that are addressable using the mask select signals.

5. The memory apparatus as in claim **1**, wherein said at least one memory subsystem includes a plurality of memory subsystems that are arranged in rows to form a plurality of multiple bit words;

wherein the first match outputs for the memory subsystems forming each multiple bit word are connected together in a wired-OR connection, the logical-OR connections for the plurality of multiple bit words outputting a second match output.

6. The memory apparatus as in claim **5**, wherein each of the plurality of multiple bit words has an associated address location, the memory apparatus further comprising:

means for outputting an address location corresponding to the second match output.

7. The memory apparatus as in claim **6**, further comprising:

means for outputting a data output associated with the address location corresponding to the second match output.

8. A method of storing ternary data, comprising the steps of:

initializing a binary-to-ternary conversion register by storing binary-to-ternary mask data in the binary-to-ternary conversion register, wherein the binary-to-ternary mask data comprises information relating to the location of don't care bits for input data;

inputting a single bit of binary data to the binary-to-ternary conversion register;

converting the single bit of binary data into first and second ternary data outputs using the binary-to-ternary conversion register; and simultaneously storing as stored content addressable memory (CAM) data the first and second ternary data outputs in first and second memory cells forming a content addressable memory.

9. The method as in claim 8, further comprising the steps of:

providing input search binary data;

searching for a match of input search binary data to the stored CAM data; and

generating a match valid output when the input search binary data matches any of the stored CAM data.

10. The method as in claim 9, further comprising the step of:

generating an address corresponding to a location in the CAM where the match is found.

11. The method as in claim 9, wherein the step of searching for a match is further comprised of the steps of:

converting the input search binary data into ternary search data, using the binary-to-ternary conversion register; and

comparing the stored CAM data to the ternary search data.

12. A memory system comprising:

a binary-to-ternary conversion register; and

means for initializing the binary-to-ternary conversion register, comprised of:

means for storing binary-to-ternary mask data in the binary-to-ternary conversion register, said binary-to-ternary mask data comprising information relating to the location of don't care bits for input data;

a content addressable memory (CAM) comprising at least first and second memory cells for storing ternary data;

means for storing the ternary data in the CAM, comprising:

means for inputting a single bit of binary data to the conversion register;

means for converting the binary data into two bits of ternary data using the conversion register; and

means for simultaneously storing as stored CAM data the two bits of ternary data in the

first and second memory cells of the CAM.

13. The memory system as in claim 12, further comprising:

means for supplying input search binary data;

means for searching for a match of input search binary data to the stored CAM data; and

means for generating a valid match output when the input search binary data matches any of the stored CAM data.

14. The system as in claim 13, further comprising:

address generator means for generating an address corresponding to a location in the CAM where the match is found.

15. The system as in claim 13, wherein the means for searching for a match is further comprised of:

means for converting the input search binary input into ternary search data, using the conversion register; and

means for comparing the contents of the CAM to the ternary search data.

16. A memory system for providing secure asynchronous transfer mode (ATM) communications for an ATM network that transmits a plurality of data cells, each of the plurality of data cells comprising payload data and header data comprised of virtual path identifier (VPI) address data and virtual channel identifier (VCI) address data, the memory

system receiving a plurality of data input signals, encryption VPI and VCI addresses, and associate key data signals, the memory system comprising:

a ternary content addressable memory (TCAM) subsystem for storing a first one of the data input signals as stored ternary data, and for determining a match output address by comparing another one of the data input signals to the stored ternary data;

an addressable lookup table subsystem for storing the key data signals and selectively outputting the key data signals in accordance with the match output address;

wherein the TCAM subsystem and the addressable lookup table subsystem form a memory subsystem;

means for initializing the memory subsystem comprising:

means for storing the encryption VPI and VCI addresses in the TCAM subsystem;

means for storing the key data associated with the encryption VPI and VCI addresses in the lookup table;

means for separating the payload data from the header data for each of the data cells;

means for delivering the separated header data to the TCAM, wherein the TCAM selectively outputs the match output address when the separated header data matches one of the stored VPI and VCI addresses;

wherein the lookup table outputs the key data associated with the respective match output address;

means for encrypting the payload data in accordance with the key data; and

means for combining the encrypted payload data with the separated header data to form an encrypted data cell.

17. The memory system as in claim 16, further comprising means for transmitting the encrypted data cell to the ATM network.

18. A memory system for implementing a secure asynchronous transfer mode (ATM) communication system that transmits a plurality of signals comprising associated encryption key data, virtual path identifier (VPI) data, virtual channel identifier (VCI) data, and data cells, each data cell comprising payload data and header data, said header data comprised of VPI/VCI data, wherein respective ones of the encryption key data are associated with respective ones of the VPI/VCI data, the memory subsystem comprising:

a content addressable memory (CAM) subsystem for storing data therein;

an addressable lookup table;

a processor for storing VPI/VCI data into the CAM subsystem, each at a respective storage address, and for storing the respective associated encryption key data into the lookup table at a location mapped to the respective storage address;

a decoder for separating the header data from the payload data for each of the data cells;

wherein the CAM subsystem includes means for comparing the stored data therein to the separated header data to selectively determine a match address output when the separated header matches any of the stored data therein;

wherein the lookup table outputs the associated encryption key data based on the match address output;

an encryptor for encrypting the separated payload data based on the encryption key data output from the lookup table; and a combiner for combining the encrypted payload data with the separated header to form an encrypted data cell.

19. The memory system as in claim 18, further comprising:

means for communicating the encrypted data cell through standard ATM infrastructure systems.

20. The memory system as in claim 19, further comprising:

means for receiving the communicated encrypted data cell;

a decoder for separating the header data from the payload data for each of the communicated data cells;

wherein the separated communicated header data is compared to the stored data in the CAM subsystem;

wherein the CAM subsystem compares the stored data therein to the communicated separated header data to selectively determine the match output when the communicated separated header matches any of the stored data therein;

a decrypter for de-encrypting the communicated encrypted separated payload data based on the encryption key data; and

means for communicating the de-encrypted separated payload data.

21. A memory apparatus, comprising:

a plurality of memory locations each capable of storing at least one data record having multiple bit positions;

means for receiving data records from an exterior environment for storage in said plurality of memory locations;

means for storing said received data records in said plurality of memory storage locations;

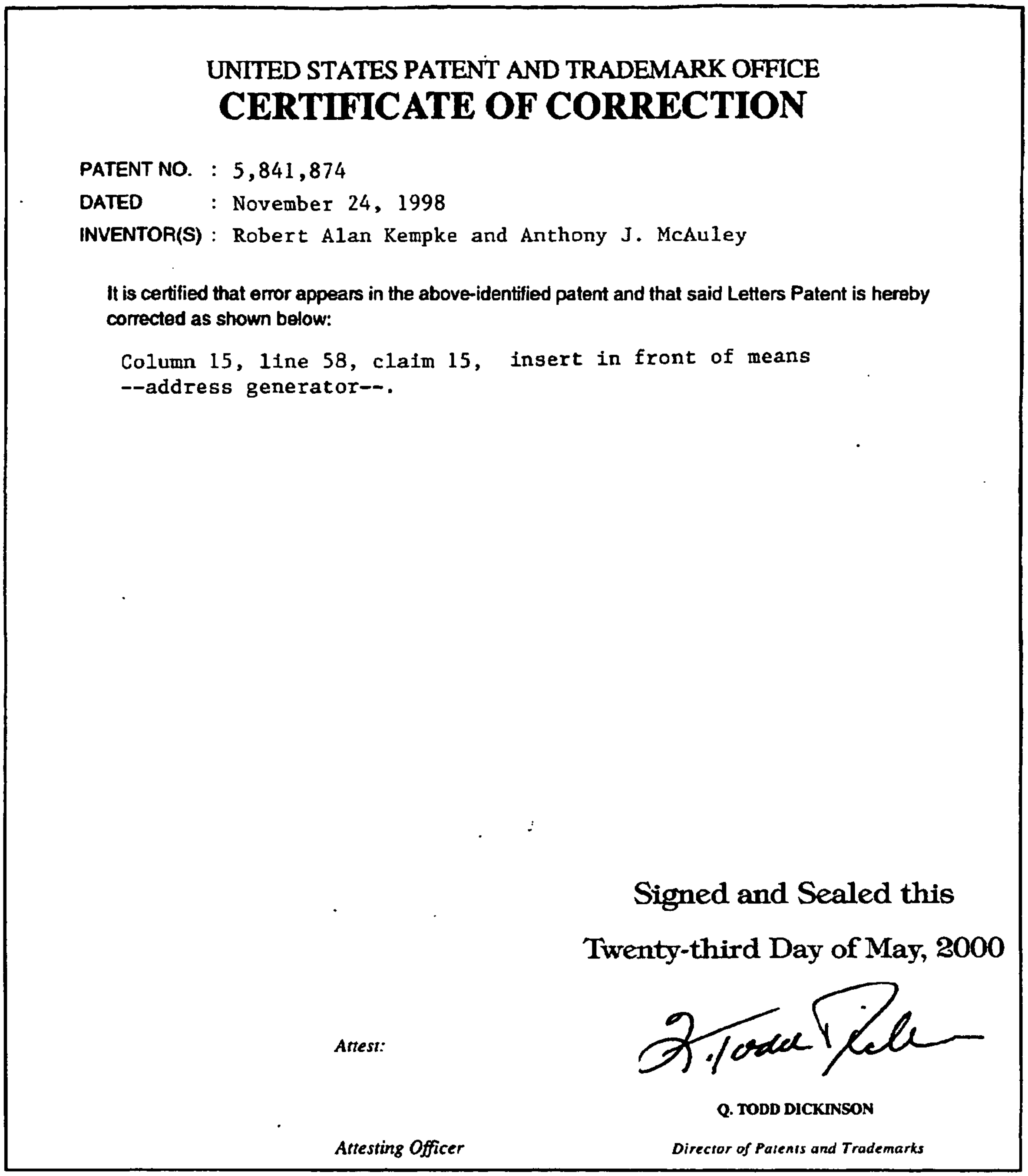
means for receiving a search data record from an exterior environment for comparison with data records stored in said plurality of memory locations;

means for comparing said search data record to multiple data records stored in said plurality of memory locations, to determine whether a record match exists between said search data record and one of the multiple data records, said comparing means comparing each bit position of said search data record to a corresponding bit position in each of said multiple data records;

means for storing first don't care bits in selected bit positions within said multiple data records stored in said plurality of memory locations, said first don't care bits each resulting in a bit match when compared with data bits in said search data record irrespective of the content of said search data record; and

means for adding second don't care bits to selected bit positions in said search data record, said second don't care bits each resulting in a bit match when compared to data bits in said multiple data records irrespective of the content of said multiple data records.

* * * * *



UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,841,874

DATED : November 24, 1998

INVENTOR(S) : Robert Alan Kempke and Anthony J. McAuley

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 15, line 58, claim 15, insert in front of means --address generator--.

Signed and Sealed this

Twenty-third Day of May, 2000

Attest:

Q. TODD DICKINSON

Attesting Officer

Director of Patents and Trademarks

103 TCAM

US006134135A

United States Patent [19]

Andersson

[11] **Patent Number: 6,134,135**

[45] **Date of Patent: Oct. 17, 2000**

[54] **MASK ARRANGEMENT FOR SCALABLE CAM/RAM STRUCTURES**

[75] Inventor: **Per Andersson**, Lund, Sweden

[73] Assignee: **SwitchCore, A.B.**, Sweden

[21] Appl. No.: **09/480,827**

[22] Filed: **Jan. 10, 2000**

[51] **Int. Cl.**^{7} **G11C 15/00**

[52] **U.S. Cl.** **365/49**; 395/425

[58] **Field of Search** 365/49; 395/425, 395/435

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,051,949	9/1991	Young	365/49
5,383,146	1/1995	Threewitt	365/49
5,386,413	1/1995	McAuley et al.	370/54
5,440,715	8/1995	Wyland	395/435
5,467,349	11/1995	Huey et al.	370/60.1
5,642,114	6/1997	Komoto et al.	365/49
5,706,224	1/1998	Srinivasan et al.	365/49

FOREIGN PATENT DOCUMENTS

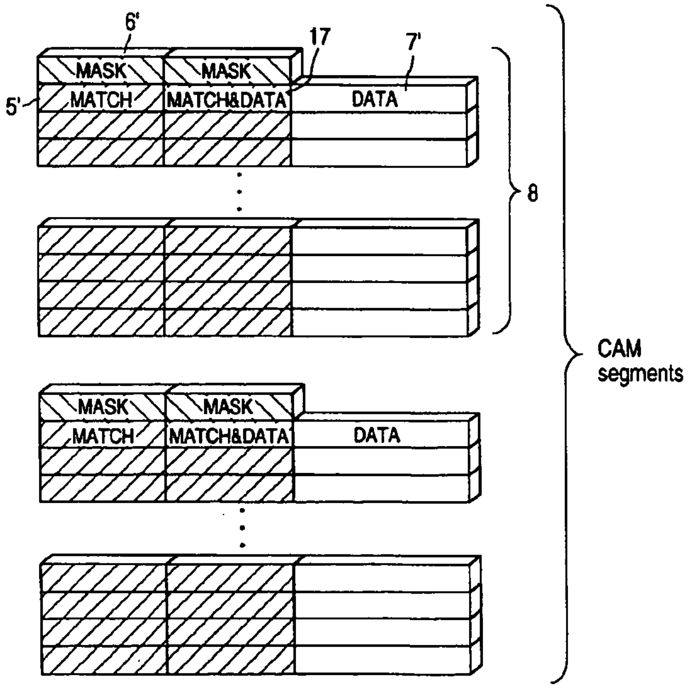
0 612 154	2/1993	European Pat. Off. .	
0 650 167 A2	10/1993	European Pat. Off. .	
0612154 A1	2/1994	European Pat. Off.	H03K 19/177
0650167 A2	10/1994	European Pat. Off.	G11C 15/00
0 883 132	5/1997	European Pat. Off. .	
0993132 A2	4/1998	European Pat. Off.	G11C 16/00
2 755 531	3/1987	France .	
2755531	3/1987	France	G11C 15/04
07/288541	4/1994	Japan .	
08/115262	10/1994	Japan .	
07-288541	10/1995	Japan	H04L 12/46
08-115262	5/1996	Japan	G06F 12/10
10/070573	8/1996	Japan .	
10/223778	2/1997	Japan .	
10-070573	3/1998	Japan	H04L 12/56
10-233778	9/1998	Japan	G06F 13/00

Primary Examiner—Vu A. Le
Attorney, Agent, or Firm—Coudert Brothers

[57] **ABSTRACT**

The invention relates to a CAM/RAM memory device with a scalable and flexible structure. The device has a number of rows of memory cells. At least one address decoder is connected by word lines to the cells of the rows. Vertical bit lines for match data implement CAM functionality of the memory device. According to the invention a mask is implemented in a row of the memory cells, the mask affecting the match data on the bit lines. Preferably, the memory device is divided into segments with a mask at the top of each segment. By means of the present invention, masking is obtained by inserting masks as mask rows between the CAM rows. The mask rows are programmed by writing the mask row cells in the same way as the CAM cells. The mask rows operate directly on the bit lines for the whole underlying segment of rows. By means of this arrangement, the invention makes efficient use of the available silicon area. The memory device has a useful application as a device for handling address look-up, e.g. in a switch or router.

15 Claims, 3 Drawing Sheets



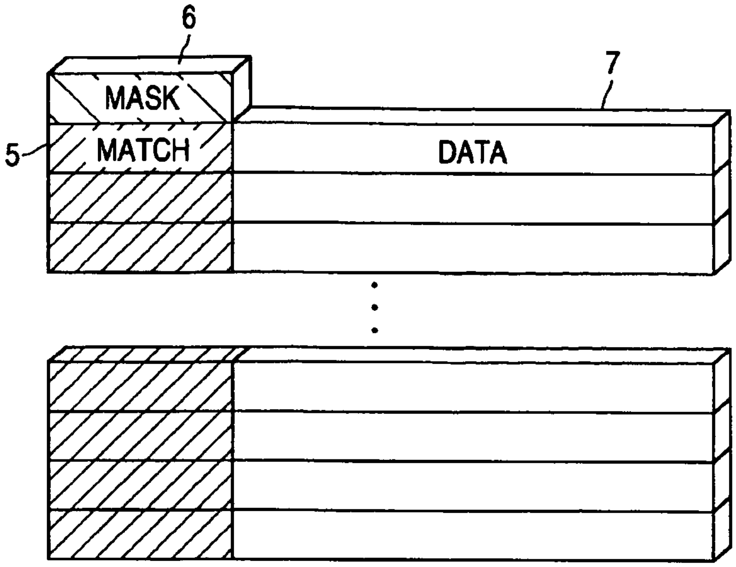


FIG. 1
(PRIOR ART)

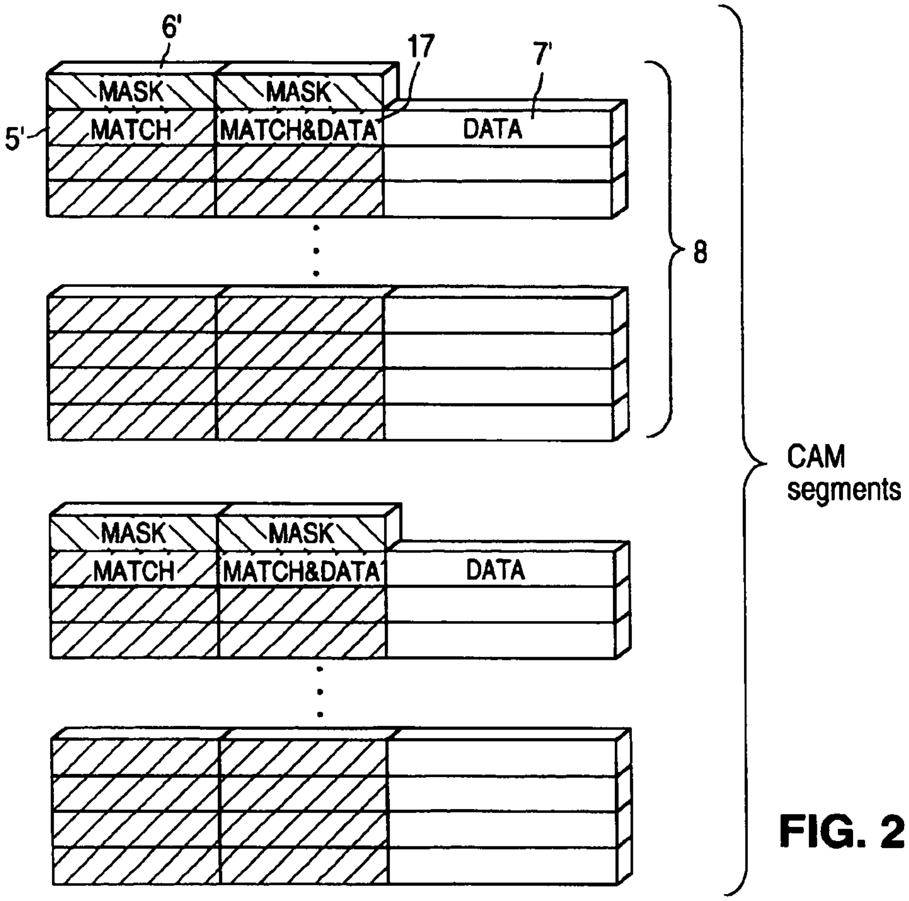


FIG. 2

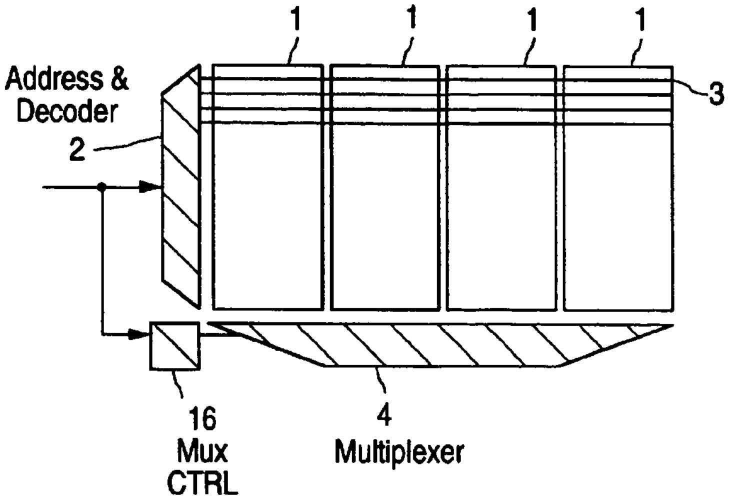

1
1
1
1
Address &
Decoder
2
3
16
Mux
CTRL
4
Multiplexer

FIG. 3

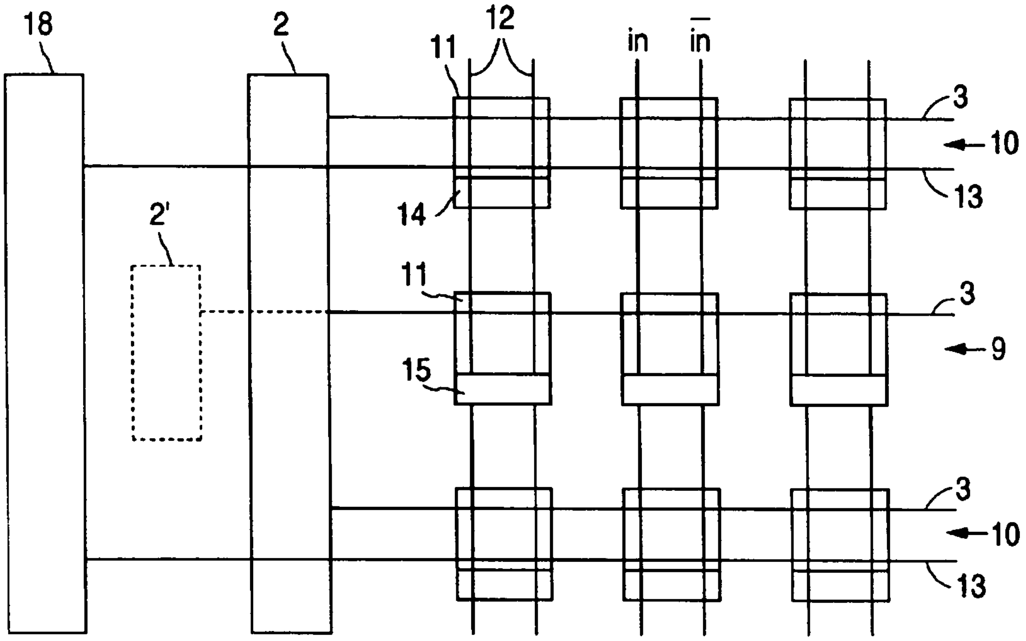

18
2
11
12
in
$\overline{in}$
3
10
13
14
2'
11
3
9
15
3
10
13

FIG. 4

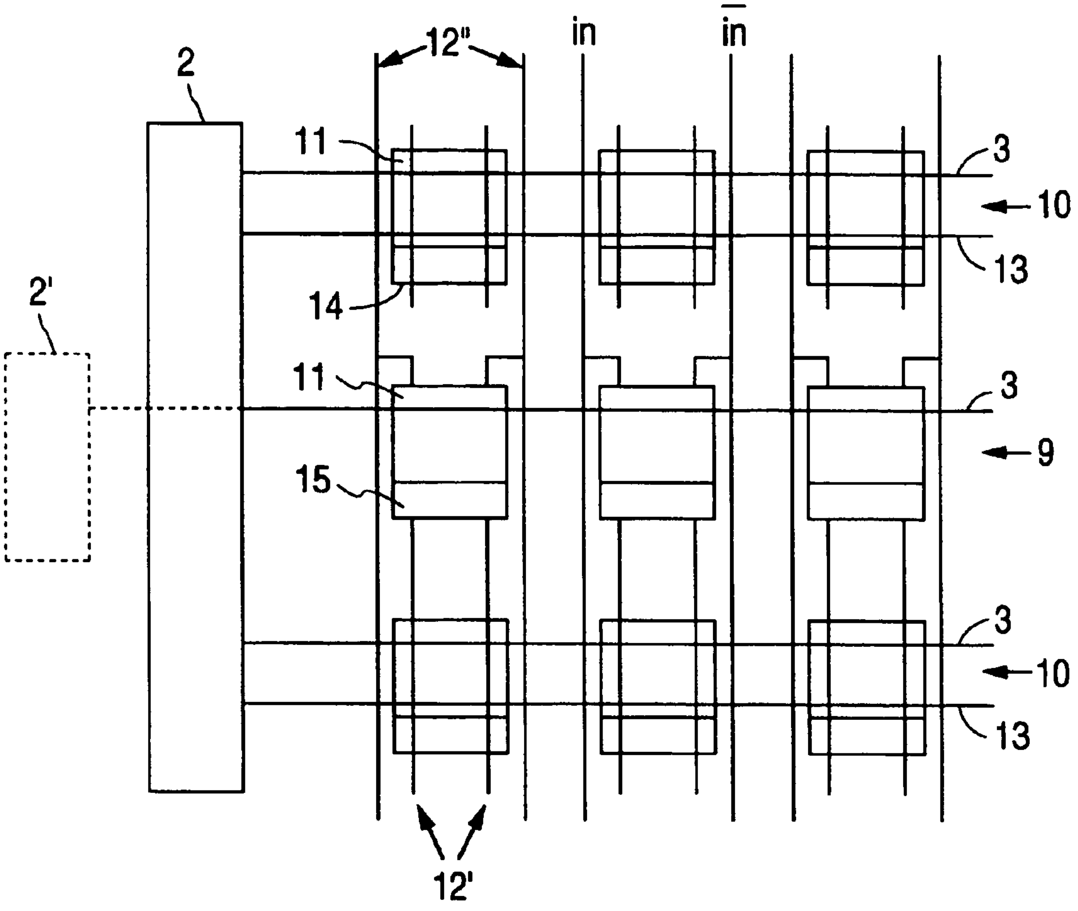


FIG. 5

MASK ARRANGEMENT FOR SCALABLE CAM/RAM STRUCTURES

FIELD OF INVENTION

The present invention relates to a CAM/RAM memory device with a scalable and flexible structure. CAM/RAM in this document means that the memory device functions both as a CAM (Content Addressable Memory) with matching operations and also has all ordinary RAM (Random Access Memory) functionality, i.e. a memory device with a CAM match operation mode and a RAM addressed read mode, where selectable parts of the memory cells can be masked off for the CAM mode and reading and writing may be performed directly through an address decoder in the RAM addressed read mode.

By means of the present invention masking is obtained by inserting masks as mask rows between the CAM rows. The mask rows are programmed by writing the mask row cells in the same way as the CAM cells. The mask rows operate directly on the bit lines for the whole underlying segment of rows. CAM/RAM memory devices are especially useful, but not limited to, address look-up devices. The memory device may employ a block structure for an efficient use of the silicon area. Blocks are arranged in parallel with common word lines running through all the blocks and operative in both RAM read/write and CAM look-up.

STATE OF THE ART

CAM memories are attractive for use in address look-up devices, and there are many different implementations of doing so. A general structure contains a CAM memory where selectable parts can be masked off to implement different look-up schemes and get the ability to handle hierarchical addresses. To get an efficient handling of address learning and changes of the look-up table it is also desirable to have a CAM that in parts works as an ordinary RAM. For address flexibility it is also desirable to have the partitions into CAM and pure RAM configurable. There exists a number of different ways of achieving these memories and examples include those described in e.g. U.S. Pat. No. 5,383,146 (Threewitt) and U.S. Pat. No. 5,706,224 (Srinvasan et al.).

When using a CAM/RAM structure as a part of an integrated device, the form factor and size are very important parameters. This is the case of a pure CAM chip as well but more so in the case of integration into single chip devices. This is important to consider when deciding how to implement the masks desired for masking out bits not to be taken into account in the match operations. The common way of implementing the masks are either to have separate mask registers as in the above U.S. patents, or to implement a mask into each memory row. Implementing a mask into each memory row (so called ternary CAM cells) is really not desirable for these kinds of area sensitive implementations, unless the application itself demands a mask for every row, since it increases the size of all CAM cells. The implementation with mask registers on the other hand requires extra logic and addressing for writing to these registers, and it is also not suitable for implementing more than a few mask rows. So, the problem is to find a way of efficiently implementing a structure where a rather small subset of the memory array shares a mask, e.g. for every 15 CAM rows there is one mask row.

The present invention solves the problem of enabling flexible and area-efficient mask implementations by inserting the masks between CAM rows in the memory array and masking the data in the bit lines. This gives a more flexible structure than the common way of placing the mask register and masking function before match data affects the bit lines as shown in FIG. 1. This also allows fine granularity and efficient integration into the memory array. Also, since no change is made to the CAM cells, these do not suffer from increased area.

By implementing a number of the memory rows as mask rows only, instead of CAM rows, it is possible to address these in the same way, and with the same address decoder as the rest of the memory. It gives a flexible way of implementing different mask granularities for different CAM implementations when designing a system.

SUMMARY OF THE INVENTION

The present invention provides a CAM/RAM memory device with a scalable structure comprising a memory having a number of rows of memory cells. At least one address decoder is connected by word lines to the cells of the rows. Vertical bit lines for match data implement CAM functionality of the memory device.

According to the invention a mask is implemented in a row of the memory cells, the mask affecting the match data on the bit lines.

In a preferred embodiment, the memory device is divided into segments with a mask row at the top of each segment.

The scope of the invention is defined in the accompanying claims.

By means of this arrangement, the invention makes efficient use of the available silicon area. Also, the memory device is flexible since all the memory cells may be handled directly in RAM mode for reading and writing. Thus, the data structure is easily changed including the mask data of the CAM functionality. The memory device has a very useful application as a device for handling address lookup, e.g. in a switch or router.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be described in detail below with reference to the accompanying drawings, of which:

FIG. 1 is a diagram of the prior art CAM/RAM structure,

FIG. 2 is a diagram of the CAM/RAM segmented structure according to the invention,

FIG. 3 is a diagram of the CAM/RAM block structure according to the invention,

FIG. 4 is a diagram of a detail of a CAM memory including CAM rows and a mask row according to a first embodiment, and

FIG. 5 is a diagram of a detail of a CAM memory including CAM rows and a mask row according to a second embodiment.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

When using these CAM/RAM structures as parts of a custom integrated single chip device, it is important that the structure is flexible both in terms of configurability and physical form factor. The intention of the invention is to enable a more flexible partitioning of the complete structure for this kind of memory.

The basic CAM/RAM structure according to the prior art is shown in FIG. 1. Input data is compared with the contents of the match field 5 and bits not to be considered in the comparison are defined in the mask 6. The data to be retrieved is stored in the data field 7. FIG. 1 illustrates one example of the prior art in which the mask is stored in a separate register and applied to the match data before the match data is applied to the CAM memory. As mentioned in the introduction, an alternative (not shown) is to implement a mask in each row of the CAM memory.

The additions needed for handling different types of address look-ups are to have a dynamically changeable configuration of the sizes of the CAM and RAM parts, respectively. In the case of IP address look-up there are a number of different address types to have support for, such as CIDR, MAC, IP-multi and VLAN. The distribution between these different types is not known in advance since it depends on the kind of device the chip is used in, and on the environment in which this device is used. The size of the necessary memory is on the other hand not that dependent on which type of address types it has to handle. A certain size of switch needs a certain amount of address memory depending on the amount of traffic it is able to handle, but this traffic can demand any of the address types, maybe all, maybe just one, but the amount is fairly stable. Therefore it is important that this partitioning of the memory can be made dynamically and in rather fine granularity.

The granularity chosen in this example is 16 word blocks where the first row in every block is a mask row. The resulting structure is shown in FIG. 2. The priority encoding for the structure of FIG. 2 is strictly based on order and therefore adds to the demand of an efficient RAM handling for moving blocks of data when reconfiguring the look-up table.

As is shown in FIG. 2, the memory comprises a number of segments 8, each segment containing a mask row 6' at the top and underlying CAM rows. Each CAM row comprises a match field 5' and a data field 7' and preferably also a part containing optionally match or data 17. Thus, it is possible to have variable length match field and the mask 6' is adapted accordingly.

For practical reasons, all segments 8 have the same size, e.g. 16 rows, having one mask row and 15 CAM rows. However, the segments 8 could be customized to various sizes.

As is known in the art, each row of the memory contains cells which hold a match part to be compared with a mask and a data part in which is the stored useful data. The configuration of the sizes of the match and data parts is dynamically changeable which is useful for handling e.g. different types of address look-ups. It is especially useful if some part of the row may be configurable to hold either match bits or data bits as the match&data field 17.

As is described in the introduction of the specification, the form factor and size is important leading to the block structure as is shown in FIG. 3. The memory is partitioned into blocks 1 of e.g. 512 words. In the figures only four blocks 1 are shown but generally the device includes more parallel blocks to offer a complete memory space of e.g. 8 k words. An address decoder 2 is connected to all the memory rows of the blocks by means of word lines 3 of which only the top four are indicated. The address decoder 2 cooperates with a multiplexer 4 to enable reading and writing of the individual memory cells. The multiplexer 4 selects output data from a sense amplifier (not shown) from one selected memory block 1. The block selection is controlled by control logic 16 (mux CTRL) which is responsive to the kind of operation (RAM addressed read or CAM match operation read), some address bits, and prioritising between hits in different blocks 1. The control logic 16 receives commands from an operation line and match hit bus (not shown).

FIG. 4 shows a detail of a block of a CAM memory and shows the masked part of the rows including two CAM rows 10 and a mask row 9. The top CAM row is the bottom row of an overlying segment and the other CAM row is the first CAM row of the next segment under the mask row 9. All rows contain memory cells 11 which may be addressed through word lines 3 by the address decoder 2. The memory cells of the CAM rows 10 are each connected to a comparator 14. As is known in the art, match data is applied to the bit lines 12, two bit lines for each cell. If the match data is not masked, one bit line of the pair is the inverse of the other, and the comparator 14 compares the match data with the contents of the memory cell 11. If there is a hit, this is signalled on the match line 13 running through the row. On the other hand, if the match data is masked, both bit lines in the pair carry zeros resulting in a hit independent of the contents of the memory cell 11 ("don't care").

The mask row 9 also contains memory cells 11, in this case for storing a mask. Each memory cell of the mask row is connected to a mask unit 15 which performs the mask function. If the match data received by the pair of bit lines is to be masked, the mask unit 15 changes the match data to a pair of zeros. If the mask is not to be applied in the cell, the match data is propagated unchanged. As is known to a person skilled in the art, there are various ways to implement the function of the mask unit 15.

In the design shown in FIG. 4, the mask rows 9 (of which only one is shown), are connected in series. Hence, it will be appreciated that a mask row affects all underlying rows of the CAM memory block. Thus, the mask rows have to be arranged in a hierarchical order in accordance with the priority function mentioned below. This is not a great disadvantage because of the hierarchical nature of IP addresses where it is desired to obtain hits with the longest prefix match.

As an alternative to a common address decoder for all rows, it may be practical to use separate address decoders for CAM rows 10 and mask rows 9. A separate address decoder 2' for the mask rows 9 is outlined in FIG. 4. This increases the size a little but might in some cases simplify the use of the CAM memory.

In FIG. 5, an alternative embodiment of the mask row arrangement with another design of the bit lines is shown. The same reference numerals are used for identical elements in FIGS. 4 and 5. In the alternative embodiment two levels of bit lines are provided. A first lower level comprises bit line pairs 12' connecting the mask row 9 (the top row) of each segment with its underlying CAM rows 10. A second higher level comprises a global bit line pair 12" running uninterrupted through all segments but only connected to the input to the top row of each segment, that is the mask row 9. These bit line pairs 12" connect the mask rows in parallel. The function is identical with the arrangement of FIG. 4, except that the match data is applied to the overlying bit line pairs 12", so that the match data is applied in parallel to all the mask rows and propagated through the bit line pairs 12' through all segments at the same time. Thus, the simpler structure of the serial bit lines 12 of FIG. 4 is traded against a faster search operation in the parallel structure of FIG. 5. Another advantage is that the mask rows do not have to be arranged in a hierarchical order, since the mask rows do not affect each other.

In a RAM addressed read mode, the address decoder 2, 2' and the multiplexer 4 cooperate to enable reading and writing by means of the word lines and bit lines. In a CAM mode, the masks 6' are applied to the bit lines thus forming match data lines running vertically through the memory blocks. There may be several hits in one and the same block 1. As is known in the art, a priority means **18**, shown in FIG. **4** only, is connected to the match lines **13** for handling CAM matches and takes care of ensuring that only one match is generated in one block 1 of memory. In other words, the priority function selects one of the plurality of matches.

The mask arrangement according to the present invention thus makes it possible to implement the masking function and mask addressing of the memory device resulting in an efficient use of the available silicon area in a flexible way. A person skilled in the art will appreciate that the embodiment of the invention described in detail here may be varied as to form and sizes of the various parts. Terms such as vertical and horizontal are used only in a figurative sense and in relation to each other. The scope of the invention is only limited by the claims below.

What is claimed is:

1. A CAM/RAM memory device with a scalable structure, comprising:

a memory having a number of rows of memory cells;

at least one address decoder connected by word lines to the cells of the rows; and

vertical bit lines for providing match data to implement CAM functionality of the memory device, wherein said rows include at least one mask row for providing a mask affecting the match data on the bit lines.

2. A memory device in accordance with claim **1**, wherein the memory is divided into segments, each segment having a predetermined number of said rows and a mask row.

3. A memory device in accordance with claim **2**, wherein the segments have the same size.

4. A memory device in accordance with claims **1**, **2** or **3**, wherein the address decoder is connected to all of the rows in said memory in order to control reading and writing of the memory cells.

5. A memory device in accordance with claims **1**, **2**, or **3**, further including a separate address decoder connected to each said mask row for controlling reading and writing of the mask.

6. A memory device in accordance with claims **2** or **3**, wherein said mask row is adapted to permit propagation of the match data on the bit lines or to permit said match data to be changed to "don't care" in accordance with the mask.

7. A memory device in accordance with claim **2**, wherein said mask row in each of said segments is connected in series by said bit lines.

8. A memory device in accordance with claim **2** or **3**, further including additional bit lines coupled only to said mask rows, and wherein in each segment the mask row and said predetermined number of rows are coupled to said vertical bit lines.

9. A memory device in accordance with claim **1**, further including a multiplexer, wherein the memory is divided into parallel blocks, each said block having a plurality of rows of memory cells and wherein each row in a given block shares a word line with its corresponding row in the other of said blocks and said multiplexer is adapted to select said parallel blocks for reading data therefrom.

10. A memory device in accordance with claim **9**, wherein a plurality of CAM matches may occur in a given block, said memory device further including priority means for selecting only one of said matches.

11. A memory device in accordance with claim **2**, wherein said memory cells include match parts and data parts that are configureable so as to be dynamically changeable.

12. A memory device, comprising:

a plurality of segments each having a CAM/RAM structure;

said CAM/RAM structure including memory cells arranged in a predetermined plurality of rows, said memory cells in each row connected by word lines and arranged to form a plurality of columns;

at least one address decoder coupled to said word lines; and

a plurality of data lines coupling said memory cells in said columns and for applying match data to said CAM/RAM structures, wherein one of said rows in each of said segments is adapted to store a mask for masking said match data.

13. The memory device of claim **12**, wherein each of said CAM/RAM structures includes a match field dynamically adaptable to a length of said mask.

14. The memory device of claim **13**, wherein said mask is operable for masking said match data for said rows in each of said segments.

15. The memory device of claim **14**, further including means for applying in parallel said match data to all of said mask rows.

* * * * *

US005136580A

United States Patent [19]

Videlock et al.

[11] **Patent Number: 5,136,580**

[45] **Date of Patent: Aug. 4, 1992**

[54] **APPARATUS AND METHOD FOR LEARNING AND FILTERING DESTINATION AND SOURCE ADDRESSES IN A LOCAL AREA NETWORK SYSTEM**

[75] Inventors: **Gary B. Videlock**, Foxborough; **Russell C. Gocht**, North Attleboro; **AnneMarie Freitas**; **Mark J. Freitas**, both of E. Walpole, all of Mass.

[73] Assignee: **Microcom Systems, Inc.**, Wilmington, Del.

[21] Appl. No.: **524,162**

[22] Filed: **May 16, 1990**

[51] **Int. Cl.**^{5} **H04J 3/24**

[52] **U.S. Cl.** **370/60**; 370/61; 370/85.13; 370/94.1

[58] **Field of Search** 370/16, 54, 60, 60.1, 370/61, 85.1, 85.2, 85.3, 85.5, 85.9, 85.13, 85.14, 92, 93, 94.1, 94.3; 340/825.05, 825.5, 825.51, 825.52, 825.53

[56] **References Cited**

U.S. PATENT DOCUMENTS

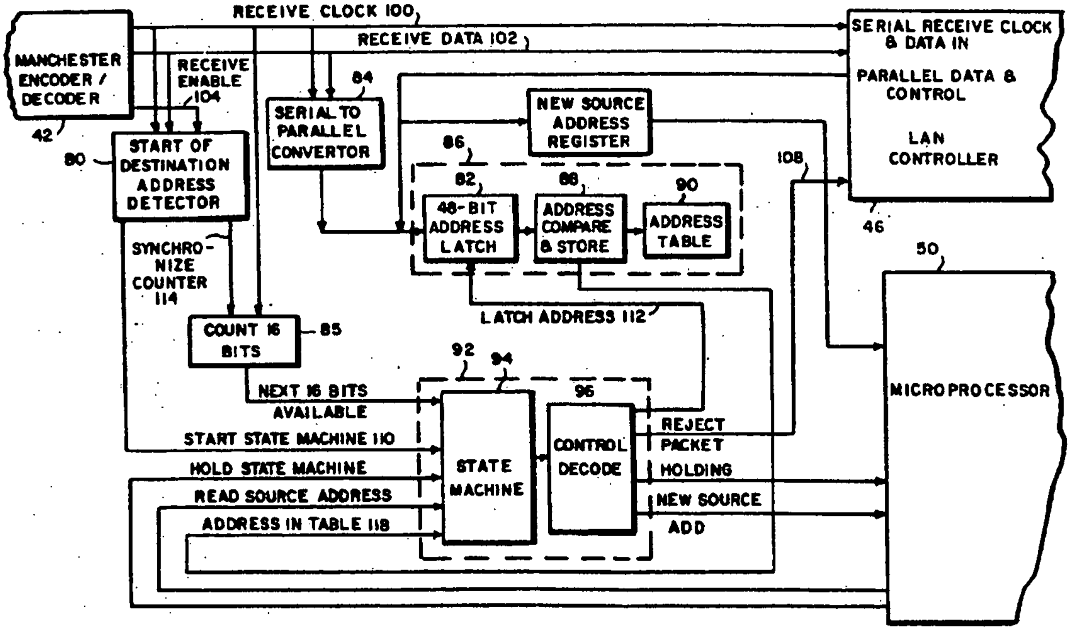
4,597,078	6/1986	Kempf	370/85.13
4,627,052	12/1986	Hoare et al.	370/85.13
4,706,081	11/1987	Hart et al.	370/61
4,715,030	12/1987	Koch et al.	370/85.13
4,831,620	5/1989	Conway et al.	370/85.13
5,027,350	6/1991	Marshall	370/85.13

Primary Examiner—Douglas W. Olms
Assistant Examiner—Alpus H. Hsu
Attorney, Agent, or Firm—Schiller & Kusmer

[57] **ABSTRACT**

An apparatus and method for learning and filtering destination and source addresses in a local area network (LAN) system is provided to facilitate the transfer of information packets from one local area network to another. The apparatus provides a link between two (or more) remotely located LANS through the use of a LAN bridge. The apparatus has a LAN controller interface, a microprocessor, a state machine and one or more content addressable memories which function to monitor an information packet received from the LAN. One such apparatus is connected to each of the LANs to be interfaced and each of these apparatus to each other in order to communicate messages between the LANs. Upon detection of an information packet, the apparatus examines both the source and destination addresses of the packet. The source address is compared to a dynamically generated table of source addresses and then compared to the incoming destination address. If the destination address is found in the table of source addresses, than the packet is destined for a node on the local LAN and thus will not be sent over the bridge between the two LANs. If, on the other hand, the destination address is not in the table, the packet is forwarded to the remote LAN.

15 Claims, 6 Drawing Sheets



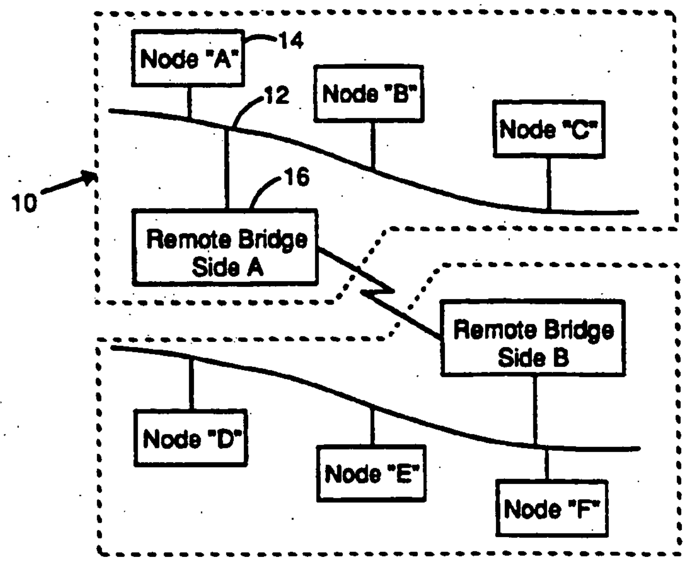


FIG.1

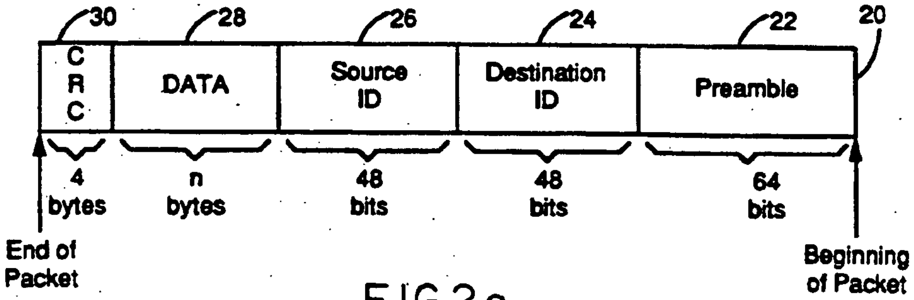


FIG.2a

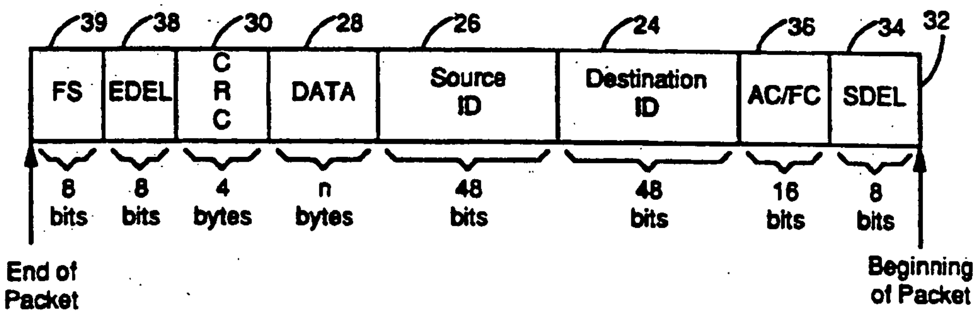


FIG.2b

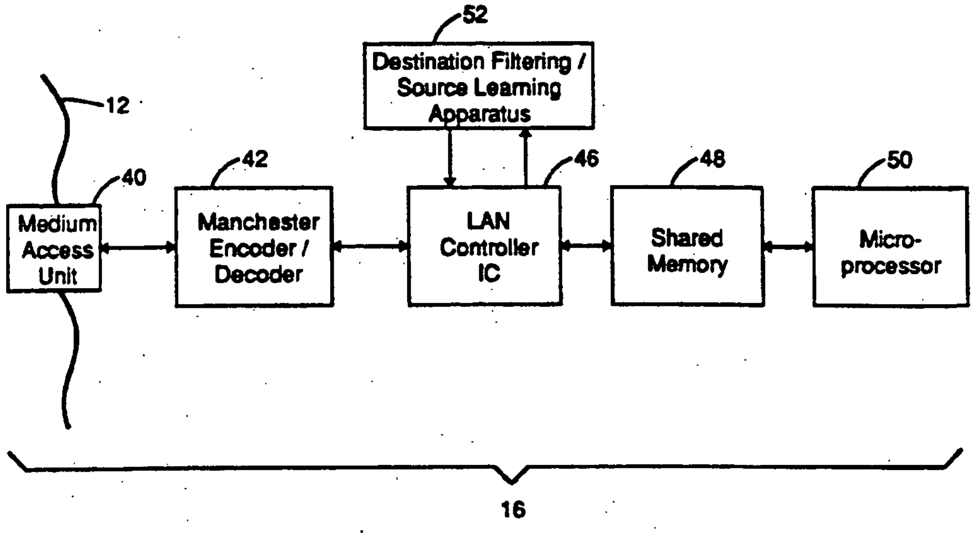

52
Destination Filtering / Source Learning Apparatus
12
40
Medium Access Unit
42
Manchester Encoder / Decoder
46
LAN Controller IC
48
Shared Memory
50
Micro-processor
16

FIG. 3

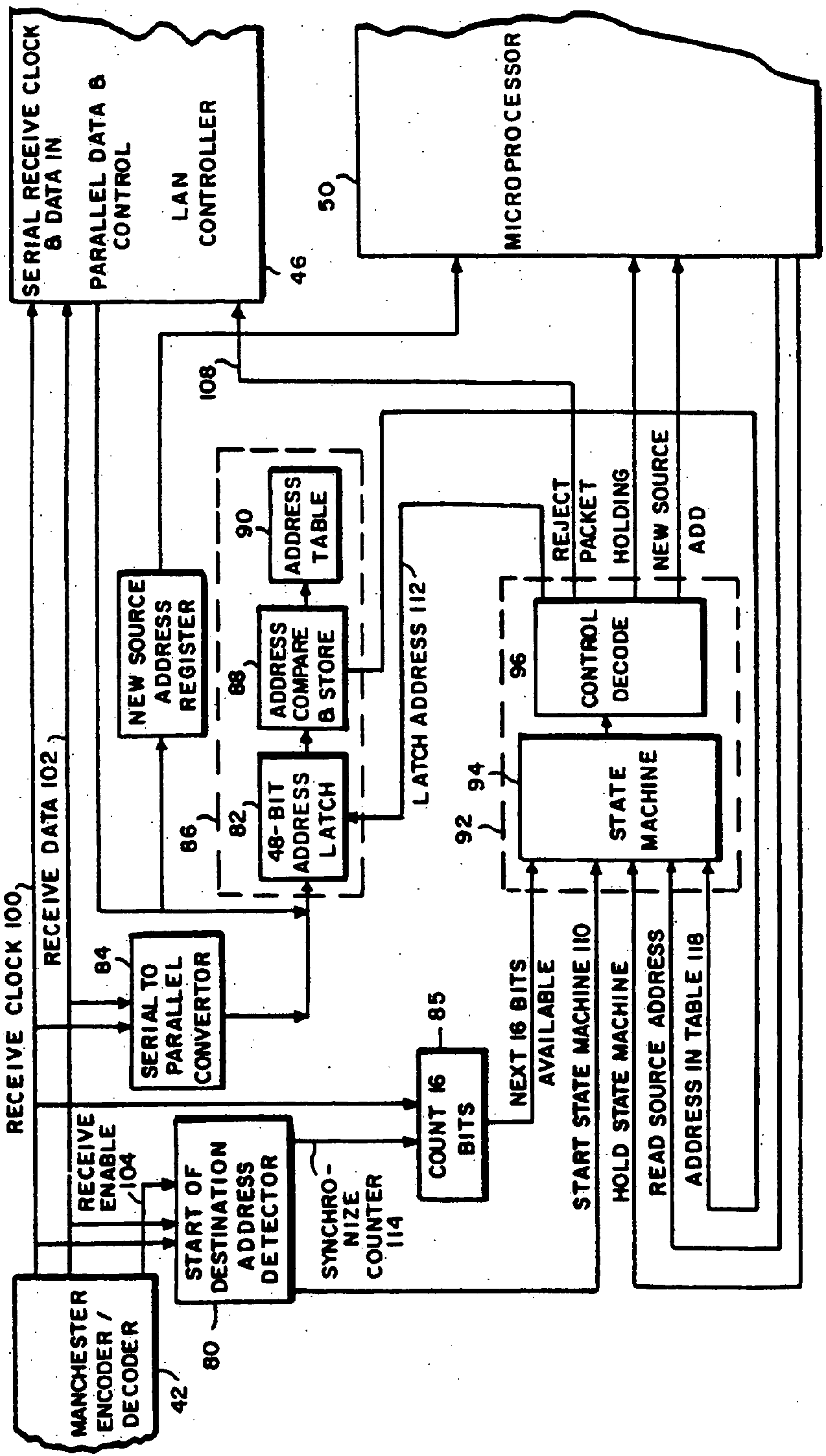

SERIAL RECEIVE CLOCK & DATA IN
PARALLEL DATA & CONTROL
LAN CONTROLLER
46
50
MICROPROCESSOR
108
NEW SOURCE ADDRESS REGISTER
90
ADDRESS TABLE
88
ADDRESS COMPARE & STORE
82
48-BIT ADDRESS LATCH
86
LATCH ADDRESS 112
REJECT PACKET
HOLDING
NEW SOURCE ADD
96
CONTROL DECODE
92
94
STATE MACHINE
RECEIVE CLOCK 100
RECEIVE DATA 102
84
SERIAL TO PARALLEL CONVERTOR
85
COUNT 16 BITS
NEXT 16 BITS AVAILABLE
START STATE MACHINE 110
HOLD STATE MACHINE
READ SOURCE ADDRESS
ADDRESS IN TABLE 118
RECEIVE ENABLE 104
START OF DESTINATION ADDRESS DETECTOR
SYNCHRONIZE COUNTER 114
80
MANCHESTER ENCODER / DECODER
42

FIG. 4

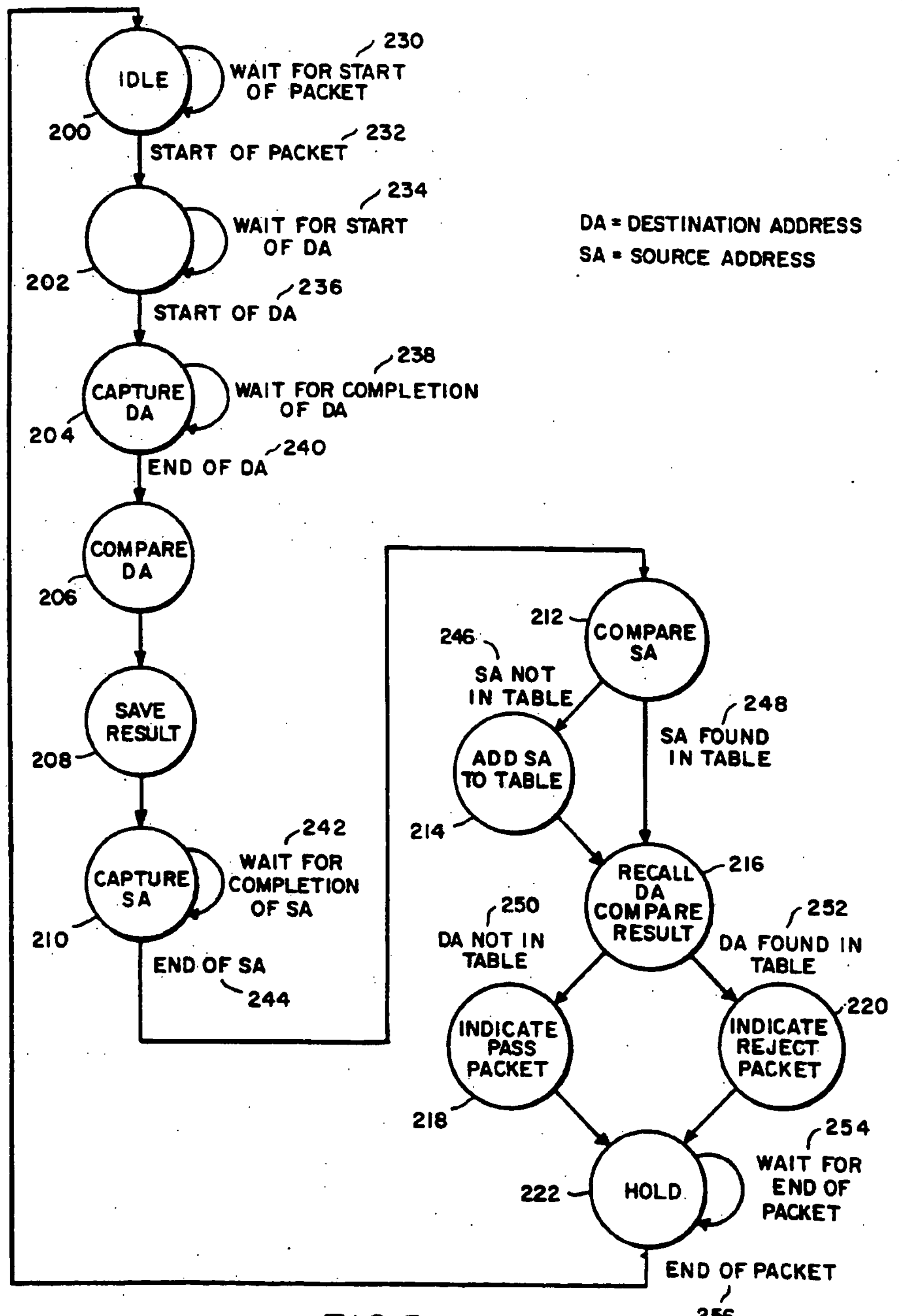

IDLE
200
WAIT FOR START OF PACKET
230
START OF PACKET
232
202
WAIT FOR START OF DA
234
START OF DA
236
CAPTURE DA
204
WAIT FOR COMPLETION OF DA
238
END OF DA
240
COMPARE DA
206
SAVE RESULT
208
CAPTURE SA
210
WAIT FOR COMPLETION OF SA
242
END OF SA
244
DA = DESTINATION ADDRESS
SA = SOURCE ADDRESS
COMPARE SA
212
SA NOT IN TABLE
246
ADD SA TO TABLE
214
SA FOUND IN TABLE
248
RECALL DA COMPARE RESULT
216
DA NOT IN TABLE
250
DA FOUND IN TABLE
252
INDICATE PASS PACKET
218
INDICATE REJECT PACKET
220
HOLD
222
WAIT FOR END OF PACKET
254
END OF PACKET
256

FIG.5

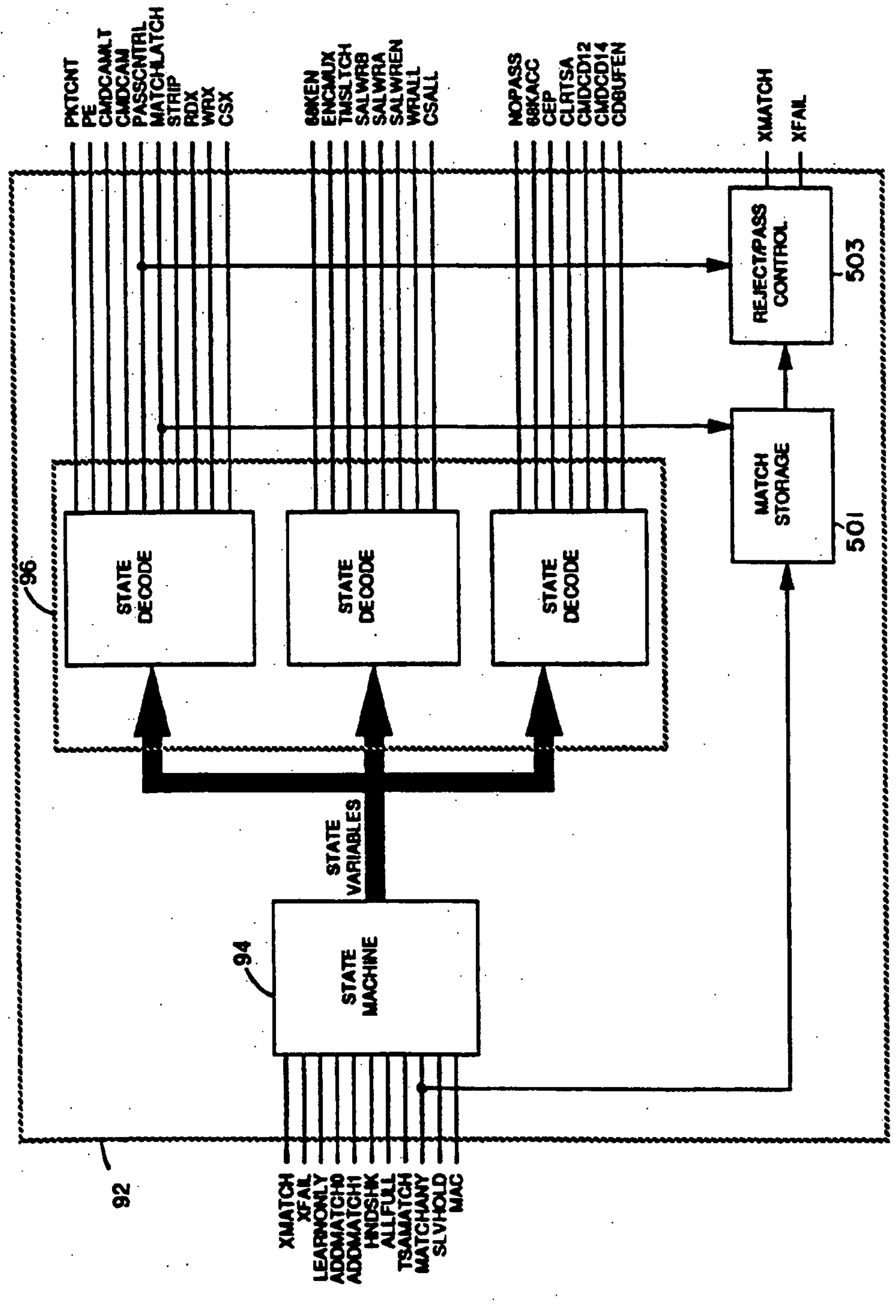

PKTCNT
PE
CMDCAMLT
CMDCAM
PASSCNTRL
MATCHLATCH
STRIP
RDX
WRX
CSX
68KEN
ENCMUX
TMSLTCH
SALWRB
SALWRA
SALWREN
WRALL
CSALL
NOPASS
68KACC
CEP
CLRTSA
CMDCD12
CMDCD14
CDBUFEN
XMATCH
XFAIL
REJECT/PASS CONTROL
503
MATCH STORAGE
501
96
STATE DECODE
STATE DECODE
STATE DECODE
STATE VARIABLES
94
STATE MACHINE
92
XMATCH
XFAIL
LEARNONLY
ADDMATCH0
ADDMATCH1
HNDSHK
ALLFULL
TSAMATCH
MATCHANY
SLVHOLD
MAC

FIG. 6

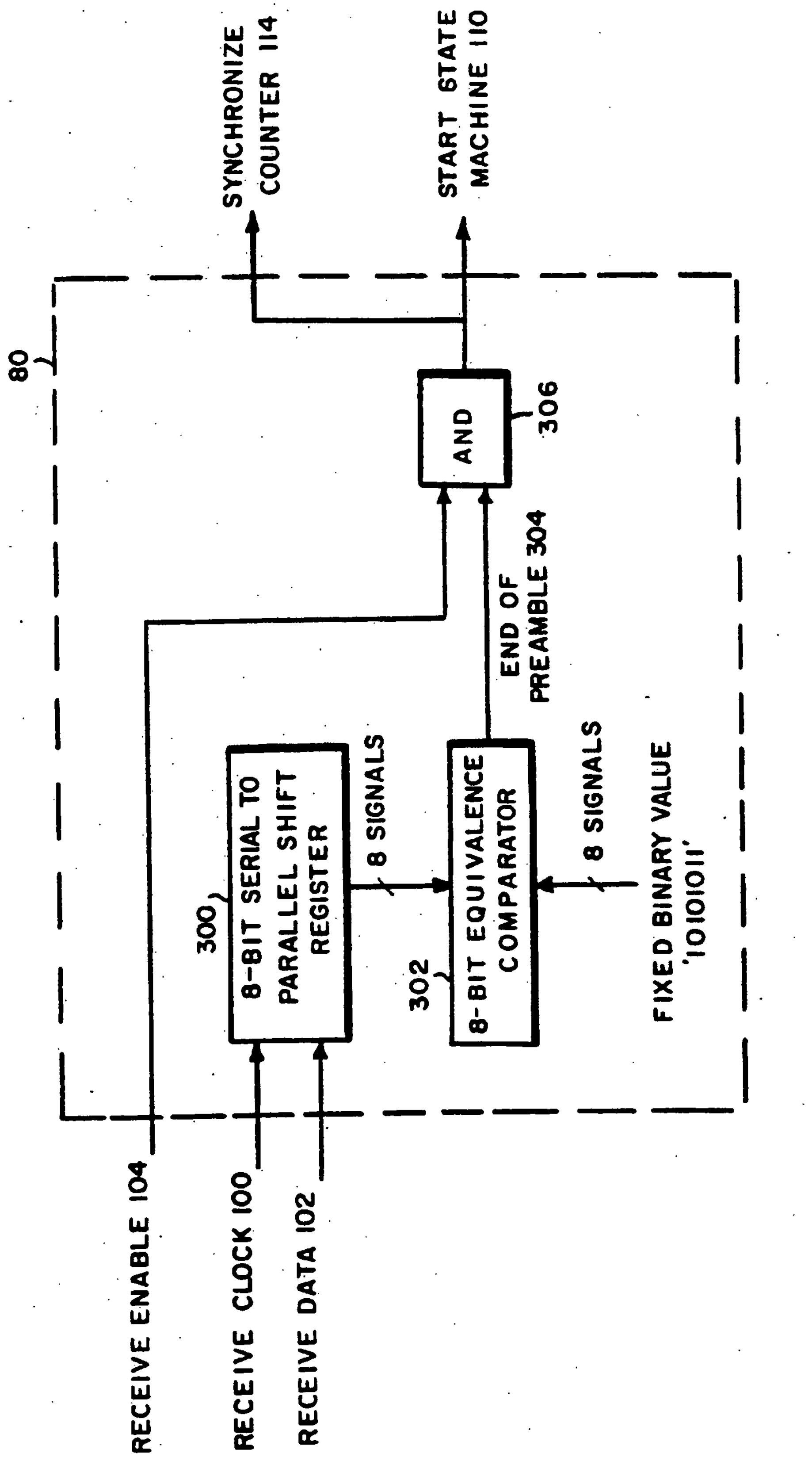

SYNCHRONIZE COUNTER 114
START STATE MACHINE 110
80
AND
306
END OF PREAMBLE 304
8-BIT SERIAL TO PARALLEL SHIFT REGISTER
300
8 SIGNALS
8-BIT EQUIVALENCE COMPARATOR
302
8 SIGNALS
FIXED BINARY VALUE '10101011'
RECEIVE ENABLE 104
RECEIVE CLOCK 100
RECEIVE DATA 102

FIG. 7

APPARATUS AND METHOD FOR LEARNING AND FILTERING DESTINATION AND SOURCE ADDRESSES IN A LOCAL AREA NETWORK SYSTEM

BACKGROUND OF THE INVENTION

This invention relates generally to Local Area Network (LAN) systems, and more particularly to a system in which two or more separate LANs may be connected or "bridged" so that traffic between LANs is restricted to those destinations which are in the remote LAN. This results in more efficient use of the processor control in the bridge device and faster throughput of messages from one LAN to another.

The computer industry has recently seen the widespread growth of LAN systems. A LAN is, broadly, a computer communications system which is shared by users in a limited (hence the terminology local) locale to share information and resources. A wide area network (WAN) is, by comparison, a computer communications system in which the users are dispersed over an unlimited geographical area (hence the terminology wide). LANs and WANs come in many varieties and types. A good description of LAN types and their respective functioning is given in *Understanding Local Area Networks*, by S. Schatt, published by Howard W. Sams & Co., 1989. WANs may take the form of a large single interconnected system or be composed of a plurality of LANs connected together so as to form a WAN. It is to the latter of these systems to which the invention of the present invention pertains.

The present invention relates to WANs composed of a number of interconnected LANs. These LANs may already have been in existence and the user may have a need to connect them in some fashion because, for example, the company has opened a new office with its own LAN system to which the home office must communicate. The company has the choice of developing a new WAN system so that both offices may be connected or may utilize the existing separate LANs and provide some means to allow the two systems to communicate. Also, the company may have a preference for a particular type or brand of LAN and a wish to remain with that system. Additionally, the needs of a particular user will depend on the functionality demanded. For example, a user may merely have a need to transfer batch information from on LAN to another on an occasional basis. On the other hand, the user may have a need to have a continual transfer of information between the LANs and the need to have this transfer take place seamlessly. It is to the latter functionality that the device of the present invention pertains. Each of the LANs usually contains a plurality of stations or nodes, which can both receive and transmit information to other stations and nodes.

Two of the more popular LAN systems commercially available today are the Ethernet and Token Ring LAN systems. These LANs are described and specified, respectively, by IEEE Recommendations 802.3 and 802.5. They differ from one another both as to the physical structure and as to the structure of their information packets. A thorough description of the respective systems is contained in the publication *Understanding Local Area Networks* referred to above. In particular, the structure of the packets presents different problems when attempting to interface two LANs utilizing either of these types. However, in both types of LANs and in LANs other than those of the Ethernet or Token Ring type, the structure of the information packets has both a source identification (Source ID) and a destination identification (Destination ID), so that the sender node and receiver node can communicate and so that the receiver node can acknowledge to the sender node whether or not the message has been received. The present invention, although not restricted in any way to the interfacing of two LANs (both of which are either of the Ethernet or Token Ring type), allows the efficient flow of messages from one LAN to another, as will be hereinafter described.

A fundamental problem encountered by the interfacing of two LANs, either locally or remotely, is that when a station or node on one of the LANs (with a certain Source ID or address) wants to send a message to another node (with a certain Destination ID or address), the source node may not know whether the destination node is located on its own LAN or on the other LAN. Thus, in certain circumstances, the bridging circuitry may automatically send all information packets to the remote LAN, even if the destination node is not on the other LAN. This takes up valuable processor time in the processor contained in the bridging circuitry. As a result, the throughput from one LAN to the other over the link between the two LANs may decrease, leading to inefficiency in the system.

Other than simply ignoring the fact that it may not be known whether the destination node is on one LAN or another, it is possible to provide a method by way of a software solution under which not all packets are sent to the remote LAN. Under this solution, all destination ID comparisons are accomplished in software, but only after the complete packet is received. An address table is built into the software of the bridge system to learn the source addresses. Unfortunately, due to the nature of the low level Token Ring protocol, the Address Recognized/Frame Copied indicator bits, which appear at the end of the packet after the Ending Delimiter, must be set in real time. A description of the foregoing structure is given in *Token Ring Access Method and Physical Layer Specifications*, published by the IEEE, 1985, pages 27 39, as well as in the *Understanding Local Area Networks* publication referred to above, especially at pages 46-47 and 79-106. The node must know that the packet was addressed to the node and that there is a buffer available. This is easily accomplished by the Token Ring Controller IC in a single address environment (such as a workstation or server), but in a transparent bridge environment, where every packet must be looked at, this can only be accomplished in hardware. It can be ignored by the transparent bridge designer because the low level protocol is ignored by some workstation/server software developers, but there are those packages that do not ignore the protocol and will indicate fault on the network if the bits are not set properly.

It is therefore a principal object of the present invention to provide an apparatus and method for learning and filtering destination and source addresses in a local area network system.

It is another object of the present invention to improve throughput for communications from one LAN to another LAN over the device which embodies the present invention by using the processor only for information packets which will be forwarded to the remote LAN.

It is yet another object of this invention to facilitate communication between two LANs of the same type remotely located regardless of whether the LANs are of the Ethernet or the Token Ring variety.

SUMMARY OF THE INVENTION

In accordance with the present invention, when bridging data between two or more Local Area Networks (LANs) not all data must be exchanged. The embodiment of the present invention will learn the station addresses of the nodes attached to the bridge, and then filter out packets that are destined for these local nodes, thereby relieving link congestion.

The invention is composed of a state machine, which may be in the form of a programmable logic array (PLA), a microprocessor interface, a LAN controller chip interface, and one or more content addressable memories. The state machine monitors the LAN controller chip to determine when a new packet is being received from the LAN. It then monitors the LAN controller chip interface to evaluate the destination and source addresses. The source address is compared to a table of previously learned source addresses and added to the table if it is not already in the table. The table of source addresses is compared to the incoming destination address. If the destination address is found in the table of source addresses, then the packet is being sent to a node on the local LAN and should not be sent through the bridge. If the destination address is not found in the table, then the packet should be forwarded over the link to the other LAN.

There are various handshake bits between the state machine and the microprocessor interface. One of the bits insures that the microprocessor reads the learned source address before another source address can be learned. Another insures that the microprocessor does not access the table of source addresses unless the state machine is in a "holding" loop. The table of source addresses has a finite size. There is an indication to the microprocessor that indicates when the table is full.

This invention improves known technology by allowing the low-level Token Ring protocol Address Recognized Indicator (ARI) and Frame Copied Indicator (FCI) bits, found in the Frame Status (FS) byte that follows a packet, to be set in real time (as the packet is being relayed through the LAN receiver to the LAN transmitter). If these bits are not set correctly, then low level software may indicate LAN errors to the node's host processor, which could result in the node removing itself from the ring.

The present invention also gives the bridge, whether it be Ethernet or Token Ring, the ability to filter all LAN packets regardless of the load of data on the network. The filtering rate of the bridge is no longer a function of the performance of the microprocessor. The microprocessor will only be given packets that need to be forwarded to the remote LAN.

The present invention is facilitated by the availability of Content Addressable Memory (CAM) devices, where the incoming destination address and source address can be compared to a table of previously stored addresses simultaneously. This makes the present invention capable of comparing large numbers of addresses in real time. As an alternative to the CAM devices. A memory array (48-bits wide) could be used, but the sequential search time would limit the number of addresses that could be compared before the end of the packet is reached.

The present invention is intended to be used in a bridge requiring the characteristics of transparent bridging, wherein the existence and interconnection of a plurality of LANs is not apparent or is otherwise transparent to the user. This could be a bridge devoted to transparent bridging, or a bridge that combines transparent bridging with source routing. The present invention can also be used in a mode to provide specific destination address filtering if the source address learning is turned off and the address table is filled by the host microprocessor.

The present invention is described in the specification in an IEEE 802.5 Mbps Token Ring environment and an IEEE 802.3 Ethernet 10base5, 10base2 and 10baseT environment. It could be implemented in an IEEE 16 Mbps 802.5 Token Ring environment or a 100 Mbps FDDI environment. The invention. However is not limited to these type LANs, and can be applied to any LAN that uses packet formats similar to the IEEE specifications.

Appendix A is a listing of Boolean equations and state machine description language utilized for carrying out the apparatus and method of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a representative portion of two LANs with a remote bridge providing connectivity between them.

FIG. 2*a* is a diagram of a known Ethernet/IEEE 802.3 packet structure.

FIG. 2*b* is a diagram of a known Token Ring/IEEE 802.5 packet structure.

FIG. 3 is a block diagram of a LAN bridge station.

FIG. 4 is a block diagram of the LAN source address learning/destination address filtering apparatus.

FIG. 5 is a flow diagram of the LAN source address learning/destination address filtering apparatus state machine.

FIG. 6 is an exploded view of the state machine used to implement the embodiment of FIG. 5 showing all sampled inputs and controlled outputs.

FIG. 7 is a schematic diagram of the start of destination address detector shown in FIG. 4.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring now to FIG. 1, in LAN 10, a medium 12 interconnects a number of nodes 14 to facilitate digital communication among them. Nodes may be server devices for one or more data terminals (DTEs), computers, etc. Different nodes can represent equipment supplied by different vendors, but all nodes communicate via the LAN medium in accordance a packet type protocol.

Occasionally, it is necessary for nodes to communicate with other nodes not directly connected to the LAN medium. Of the several methods to accomplish this connection, a transparent bridge 16 is shown in FIG. 1. The invention can be applied to this type of bridge, whether remote (connected to another bridge via a Wide Area Network link) of a local (co-located in the same chassis) and can also be used in conjunction with the Token Ring Source Routing protocol.

Referring now to FIG. *2a*, representing an Ethernet data packet, each data packet **20** begins with a preamble **22** of **64** bits consisting of alternating 1s and 0s and ending with two consecutive 1s. Next a sequence of forty eight destination ID bits **24** identify the node for which the packet is intended, and a sequence of 48 source ID bits **26** identify the node which originated the packet. The n bytes of data **28** appear followed by four CRC bytes **30** at the end of the packet. The structure of the Ethernet packet described herein is not a part of the present invention, but is a data packet which is described and specified by IEEE Recommendation 802.3.

Referring now to FIG. *2b*, representing a Token Ring data packet, each data packet **32** begins with a Starting Delimiter **34** of 1 byte comprised of Manchester code violations. Manchester encoding is a known method to encode serial data and is specified in the IEEE publication referred to above on page 73. Next, a sequence of two bytes comprises the Access Control/Frame Control fields **36**. Then a sequence of forty eight destination ID bits **26** identify the node which originated the packet. The n bytes of data **28** appear followed by four CRC bytes **30** at the end of the packet. The packet is closed by an Ending Delimiter **38** of 1 byte comprised of a different Manchester code violation than the Starting Delimiter **34**. Following the Ending Demlimiter **38** and outside the CRC **30** is the Frame Status Field **39** containing the Address Recognized/Frame Copied bits. The node on the LAN can determine by the setting of Frame Status Field **39** whether the destination node is located on the LAN and whether the message was received. The functioning and structure of the Frame Status Field **39** is detailed in the IEEE publication referred to above, at pages 34–35.

The structure of the Token Ring packet described herein is not a part of the present invention, but is rather a structure which is specified and described by IEEE Recommendation 802.5, contained in the IEEE publication referred to above. These Address Recognized/Frame Copied bits convey to the sending node whether or not the destination ID **24** was recognized by a node on the LAN and whether or not the receiving node copied the frame into its memory. The decision as to how these bits are to be set must be rendered before the end of the packet is relayed through the LAN controller IC.

Referring now to FIG. **3**, the transparent bridge denoted as **16** contains a medium access unit **40** that receives all digital data signals that appear on the LAN medium and transmits onto the medium digital data signals that are generated from the bridge. Digital data signals are carried on the medium **12** in Manchester encoded form. Medium access unit **40** is connected to a Manchester encoder/decoder **42** that decodes the encoded digital data received from the medium access unit **40** into a stream of non Manchester encoded data and Manchester encodes streams of non Manchester encoded data generated by the node for delivery to the medium access unit **40**. A commercially available integrated circuit available for purposes of providing the functioning of the Manchester encoder described herein is an AMD 7992 device, available from Advanced Micro Devices.

The Manchester Encoder/Decoder determines that there is signaling on the LAN medium. When it begins decoding the signal, it conveys the decoded data and

clock to the LAN Controller **46** LAN Controller **46** may be a commercially available integrated circuit, such as the AMD 7900, available from Advanced Micro Devices. In the absence of the present invention, all packets received by the LAN controller **46** will be placed in the shared memory **48**, requiring the microprocessor **50** to allot valuable time to looking at each packet and determining whether it should pass it along to the remote bridge or discard the packet. In addition, and again in the absence of the present invention, in some Token Ring applications the low level packet protocol will be violated because the Address Recognized/Frame Copied bits in the Frame Status Field **39** could not be set while the packet was passing through the LAN Controller.

Referring now to FIG. **4**, the invention facilitates two methods to interface with the data stream being received from the LAN: (1) a serial path using the Receive Clock **100** and Receive Data **102** signals that connect the Manchester encoder/decoder and the Ethernet LAN controller chip (for Ethernet/802.3 applications); and (2) a parallel path that allows the invention to take data directly from the Token Ring Controller chip parallel bus **104** after the Manchester encoder/decoder and the internal LAN Controller serial to parallel conversion (for Token Ring/802.5 applications)

In the case of an Ethernet LAN, a separate circuit **80** looks for the Ethernet preamble **22**. The Ethernet preamble consists of 64 bits of alternating ones and zeroes, the end of which is designated by two consecutive ones (illustrated by the binary octet '10101011') The Start of Destination Address Detector **80**, shown in FIG. **4** and in detail in FIG. **7**, uses a serial to parallel shift register **300** with the receive data **102** signal feeding the serial input and the receive clock **100** continuously clocking the data through shift register **300**. Any time the parallel output of the shift register **300** matches the fixed binary pattern 10101011 (the end of-preamble octet described and specified in the IEEE 802.3 specification) of the 8 bit equivalence comparator **302**, an end of preamble signal **304** is asserted. This signal **304** is "ANDed" with the receive enable signal **104** at **306** such that the Start State Machine signal **110** and the Synchronize Counter signal **114** will assert only when a packet is being received.

The circuit **80** generates a signal **114** that synchronizes the serial to parallel shift register **84** to the incoming data stream. A counter **85** is started at this time to determine when 16-bits of the destination ID have been serially shifted. At the end of 16 bits the counter signals the state machine **94** that the parallel registers can be "dumped" to the 48-bit address latch **82** within the content addressable memory **86**. This 48-bit latch within the content addressable memory chip has to be loaded in three consecutive 16-bit operations. In the remainder of the description of the present invention, the operation of the invention is the same for Ethernet and Token Ring The serial to parallel shifting and synchronization are done independently of the state machine, so that the state machine does not need to know which type of LAN (Ethernet or Token Ring) it is operating on.

In the case of a Token Ring LAN, the Start of Destination Address Detector **80**, the Serial to Parallel Convertor **84**, and the 16 bit Counter **85** are contained within the Token Ring Chip set commercially available from Texas Instruments (as TMS380). The TMS380 presents a parallel bus that contains the information to operate the state machine.

The state machine 92 controls the address storage latches 82 with the latch address signal 112. The state machine referred to generally in FIG. 4 as 92 actually is a representation of a state machine 94 and a control decode 96, described in detail below with reference to FIG. 6. When a complete address is latched into the address storage latches 82 of the content addressable memories 86, the content addressable memories automatically begin a table comparison process. One content addressable memory will simultaneously compare 256 48-bit data string (in this case LAN addresses) and report the result as a match (if the comparing string is found in at least one of the 256 locations) or a no match (if the comparing string is not found in any of the 256 locations). One embodiment of the present invention allows for up to eight content addressable memories to be used in parallel operation for a total of up to 2048 (256 times 8) 48 bit LAN addresses. Addresses can be stored in the content addressable memories by one of two methods: (1) the microprocessor 50 (which may be a M68000 series microprocessor available from Motorola) can directly enter the LAN addresses (which can be either learned from packets received or from a predetermined table entered by a user of the LAN bridge), or (2), the state machine can learn the source addresses 26 from the incoming packets as described below. When a comparison to the address table 90 is completed, the address compare and store logic 88 signals the state machine 92 with the address-in-table 118 signal. The state machine 92 signals the LAN controller to discard a packet by asserting the reject packet 108 signal. In the case of an Ethernet LAN, this signal is "ORed" with the collision signal from the Medium Access Unit 40, to force the Ethernet LAN Controller 46 to discard the packet. In the case of Token Ring, this signal is connected to the XFAIL signal of the Token Ring Chip set (Texas Instruments TMS380), and its inverse is connected to the XMATCH signal. The assertion of XFAIL will cause the Token Ring Chip set to discard the packet unless the packet is determined by the Token Ring Chip set to be addressed to the bridge directly.

Referring now to FIGS. 4 and 5, when no packets are being received from the Manchester encoder/decoder 42, the state machine is idle at step 200. The state machine remains idling while waiting for the start of the packet at 230. As soon as the start of a packet is indicated to the apparatus of FIG. 4 at step 232, the start of destination address detector 80 begins looking for the destination ID 24 (shown in FIGS. 2*a* and 2*b*) in the data and the state machine waits at 202 for an indication of the start of the destination address at 234. When the destination address detector 80 determines that the destination ID 24 is about to start, the detector 80 signals this to the state machine 92 by asserting the start state machine signal 110 at 236. This places the state machine 92 in the capture destination address mode at step 204.

The state machine 92 then captures the destination ID 24 in the address storage latches 82. After the destination ID 24 is captured completely at 238 and completely stored at 240, the content addressable memories 86 compare at step 206 the destination ID 24 with the address table 90 of previously stored source IDs 26 (shown in FIGS. 2*a* and 2*b*). At the present time content addressable memories are available commercially, such as the AMD 99C10 integrated circuit available from Advanced Micro Devices, in size of 256 address capacity. These CAMs may be ganged together to allow 512, 1024, etc. of addressable memory. The destination ID comparison is completed before the source ID is captured. The result of the comparison is conveyed to the state machine by the address in table signal 118. The state f this signal is stored in step 208 by the state machine 92. After the comparison of the destination ID 24, the state machine enters the capture source destination mode in step 210. When the complete source ID 26 is captured in 242 and completed at 244, the state machine 92 causes the content addressable memories to compare in step 212 the stored source ID 26 to the address table 90 of previously stored source IDs. When the comparison is complete, the state machine 92 checks the state of the address-in-table signal 118. If this signal is active such that the source address is found in the table, then the address has been stored previously as denoted at 248. If this signal is not active such that the source address is not in the table at 246, then the state machine will enter the new source ID into the address table 90 by sequencing through the add source address to table mode in step 214. The stored state of the address in table signal 118, in step 216 is then used to signal to the LAN controller to either reject the packet at 220 if the destination address is found in the table at 252 using the reject packet signal 108 or pass the packet to the microprocessor in step 218 if the destination address is not found in the table denoted at 250. In the case of Ethernet, this signal is "ORed" with the collision signal from the Medium Access Unit 40 to force the Ethernet LAN Controller 46 to discard the packet. In the case of Token Ring, signal 108 is connected to the XFAIL signal of the Token Ring Chip set (Texas Instruments TMS380), and its inverse is connected to the XMATCH signal. The assertion of XFAIL will cause the Token Ring Chip set to discard the packet unless the packet is determined by the Token Ring Chip set to be addressed to the bridge directly. The state machine then enters the wait for end of packet mode denoted at 254 and waits for the end of the packet at 256 in step 222 before returning to the idle state in step 200.

Referring now to FIG. 6, the implementation of the Token Ring version of FIG. 5 uses groups of signals that are represented as a single signal in FIG. 5. In FIG. 6, it is seen that the state machine 92 corresponds to the state machine 92 of FIG. 4, and is comprised of the state machine 94, and state decode 96, which may be, as noted above, programmable logic arrays. The meaning and significance of the various signals represented in FIG. 6 are given by the Table below:

TABLE OF SIGNAL NAME ABBREVIATIONS FOR FIG. 6

PKTCNT	PacKet CouNT	Increments a 16-bit counter to keep track of packets received from the LAN
PE	Parallel load Enable	Allows the address on the Token Ring Chip set address/data bus to be latched into a set of incrementing counters.
CMDCAMLT	CoMmanD CAM LaTch	When the CAM command register is read this signal will latch the address of the first empty CAM location.
CMDCAM	CoMmanD CAM	This signal enables a CAM command to be written to the CAM control register.
PASS-CNTRL	PASSCoNTRoL	This signal asserts near the end of the state machine cycle to cause the LAN controller chip to pass or reject the current packet.
MATCH-	MATCH LATCH	This signal latches the state

-continued

TABLE OF SIGNAL NAME ABBREVIATIONS FOR FIG. 6

LATCH		of the "ANDed" MATCH signals from the CAMs after the CAMs have been loaded with the destination address.
STRIP	STRIP	This signal is used when the state machine detects an automatic reject, such as a Token Ring MAC level frame.
RDS	Read one CAM	This signal is used to single out a single CAM in a multiple CAM system to be read from.
WRX	Write one CAM	This signal is used to single out a single CAM in a multiple CAM system to be written to.
CSX	Chip Select one CAM	This signal is used to single out a single CAM in a multiple CAM system to be read or written.
68KEN	68K ENable	This signal is asserted in response to a SLVHOLD signal assertion when the state machine is in the idle state and the microprocessor requests access to the CAMS.
ENCMUX	ENCode MUtipleXer	This signal latches the number of the CAM (in a multiple CAM system) where a newly learned source address has been stored.
TMSLTCH	TM380 LaTCH	This signal is synchronized with the demultiplexing of the Token Ring Chip set address/data bus.
SALWRA SALWRB	Source Address Learned WRite A/B	These signals control the addressing of a set of register files that store the newly learned 48-bit source address.
SALWREN	Source Address Learned ENable	This signal controls the writing of the newly learned 48-bit source address to a set of register files, that can subsequently be read by the microprocessor.
WRALL CSALL	WRite ALL Chip Select ALL	These signals are asserted to cause a simultaneous write to all of the CAMS. They are used to write the source and destination addresses to all of the CAMs (in a multiple CAM system) at the same time.
NOPASS	NO PASS	This signal has an identical function to STRIP.
68KACC	68K ACCess	This signal has an identical function to 68KEN.
CEP	Count Enable Parallel	This signal allows the 16-bit counter that contains the address latched from the Token Ring Chip set address/data bus (by the signal PE described above) to increment to the next sequential address, which will indicate where the next received 16-bits of the incoming packet will be stored by the Token Ring Chip set.
CLRTSA	CLeaR Transmit Strip	This signal clears the register that holds the source address of a packet that the Token Ring Chip set has queued for transmit. This prevents the state machine from learning a packet sent around the ring by this node (the bridge).
CMDCD12 CMDCD14	CoMmand CAM Data bit 12/14	These signals are used to change bits on the data bus (through an octal buffer controlled by the CMDCAM signal described above) when the state machine needs to reset the CAM segment counter bits or needs to write the source address into the CAM memory or the microprocessor data bus buffers have access to the CAM data bus.
CDBUFEN	CAM Data bus BUFfer ENable	
XMATCH	eXternal MATCH	This signal is bi-directional from the Token Ring Chip set. As an input, when both it and the XFAIL lines are active, the chip set is signalling the beginning of anew receive packet. As an output, it indicates to the chip set that it should pass the packet to the microprocessor.
XFAIL	eXternal FAIL	This signal is bi-directional from the Token Ring Chip set. As an input, when both it and the XMATCH lines are active, the chip set is signalling the beginning of a new receive packet. As an output, it indicates to the chip set that it should reject the packet currently being received, without informing the microprocessor.
LEARNONLY	LEARN ONLY	This signal is set by the microprocessor to allow the state machine to learn the source addresses of incoming packets and reject them regardless of the destination address.
ADDMATCHO ADDMATCH1	ADDress MATCH 0/1	These signals are the equivalence outputs of standard octal comparators that compare the current address on the Token Ring chip set bus with the stored and incremented address latched by the PE signal (described above).
HNDSHK	HaNDSHake	This signal indicates to the state machine that the microprocessor has read the previously learned source address. If this bit is not active, then the state machine will proceed to learn a new source address. If this bit is active, then the state machine will not learn any other source addresses, but it will still filter based on the destination address.
ALLFULL	ALL FULL	This signal indicates that the CAM(s) are full and that no more new source addresses can be stored in the CAM(s).
TSAMATCH	Transmit Strip Address MATCH	This signal is the equivalence output of a standard comparator that compares the packet currently being received with the source address of the packet most recently transmitted. This prevents the state machine from learning the source address of the packet it just transmitted.

-continued

TABLE OF SIGNAL NAME ABBREVIATIONS FOR FIG. 6		
MATCHANY	MATCH ANY	This signal is the "ORed" signal of all of the CAMs MATCH signals that indicate that a match for the destination or the source address was found one of the CAMs.
SLVHOLD	SLaVe HOLD	This signal is a request from the microprocessor to place the state machine in a "holding" loop so that the microprocessor can access the CAMs without interruption.
MAC	MAC	This signal is generated when the AC/FC portion of the Token Ring packet is present on the Token Ring Chip set data bus. It is used by the state machine to reject MAC frames.

In operation, the Start of Packet **230** is signified to the state machine **94** when both XFAIL and XMATCH are asserted. The state decoder **96** de-asserts PE and asserts CEP to capture the TMS380 bus address of the packet as it is being stored to memory. The state machine then enters the Wait for Destination Address loop in step **202**. When the next sequential address appears on the TMS3870 memory bus, the two signal ADDMATCH0 and ADDMATCH1 will assert. The state decoder asserts WRALL to the content addressable memory(ies) so they can latch a 16 bit piece of the 48 bit destination address. Then CEP is asserted to allow the next sequential TMS380 memory address to be located. This cycle occurs two more times, allowing the capture of all 48 bits of the destination address.

After the third write to the content addressable memory(ies) they will automatically begin the Compare Destination Address cycle in step **206**. Then the state machine will advance to the Save Result cycle in step **208** and the state decode will assert the MATCHLATCH signal to take a snapshot of the result of the comparison and write that result to a storage element **501** for later use. MATCHLATCH is used as the clock input to a D type flip flop, the standard storage element denoted at **501** as MATCH STORAGE. The MATCHANY signal, also known as the Address in Table denoted at **118** in FIG. **4**, will be asserted by any content addressable memory that locates the 48-bit destination address in its table. The MATCHANY signal is used as the "D" (or data) input to the storage element or flip flop **501**. The "Q" or output of the element or flip-flop **501** is sent to the Reject/Pass control element or decoder **503**.

The state machine checks the state of the ALLFULL signal to determine if there is space available in any of the content addressable memory(ies) in the case the source address (which will be captured next) needs to be stored in the content addressable memory. It also checks the HNDSHK signal to determine if the microprocessor **50** has read a previously learned source address. If either of these signals is asserted then the state machine skips the source address portion of the state machine (in steps **210**, **212** and **214**) and proceeds to recall the result of the Destination Address comparison **216**.

If both ALLFULL and HNDSHK are de asserted, the state machine continues to look for data being stored in sequential memory locations on the TMS380 memory bus. It captures the three 16 bit pieces of the Source Address in the content addressable memory(ies) address latch by asserting WRALL and CEP in the same manner as described previously. The state decode also asserts SALWRA, SALWRB, and SALWREN to capture the source address to a 48-bit register that can be read by the microprocessor if the address is added to the content addressable memory address table **90**. When the 48-bit address is complete, the content addressable memory(ies) automatically compare the address to the addresses previously stored in the address table **90** (as internal structure of the content addressable memory). The state machine advances to the Compare Source Address cycle in step **212**.

If the content addressable memory(ies) finds a match, the MATCHANY signal will be asserted to the state machine **94** and to the storage latch **501** denoted as MATCH STORAGE in FIG. **6** for later use. This signal is coupled with the TSAMATCH signal which, if asserted, will indicate that this packet currently being received was transmitted by this bridge (in a ring, the transmitter will receive all packets that it transmits and is responsible to strip off its transmitted packet so that it does not continue around the ring indefinitely). The TSAMATCH signal is created by comparing the received source address to the last queued transmit address read from the TMS380 memory bus. If the TSAMATCH signal is asserted, then the state machine does not learn the received source address (if it did learn the address of a packet that came from another LAN via this bridge, then packets originated by local nodes destined for this address will be filtered in the future, thereby defeating the bridging function, since the learned address table is used for filtering destination addresses).

If the source address is to be added to the table, then the state machine enters the Add Source Address To Table cycle in step **214** and the state decode asserts RDX, WRX, and CSX to the content addressable memory with an empty location (determined by the individual FULL signals supplied by the content addressable memory fed through a priority encoder such as the TTL chip 74147). First the content addressable memory is read with the RDX and CSX signals. The first available location is from the status register of the content addressable memory into an 8-bit register using the CMDCAMLT signal (this address indicates one of **256** memory locations). Then a command is issued to the content addressable memory by asserting the WRX, CSX, CMDCAM, and CMDCD12 signals. The state machine then advances to the Recall Destination Address Compare Result cycle in step **216**.

This group of signals consists of PASSCNTRL, STRIP, and NOPASS. All three signals control a decoder **503** denoted as REJECT/PASS CONTROL in FIG. **6** to enable either XMATCH or XFAIL (which have been held in a high impedance state). If PASSCNTRL is asserted, then the decoder **503** looks at the state of the signal stored by the signal MATCHLATCH back in the Save Result cycle **208**. If this signal is asserted, then a match was found in the content addressable memory indicating that the address was learned from a local node. In this case, since filtering is based on the destination address, the packet currently being received is destined for a local node and should not be passed to the microprocessor 50 and XFAIL will be asserted. If the latched signal is de asserted, then this packet is bound for a non local destination and should be passed to the microprocessor and XMATCH will be asserted. If STRIP is asserted, then the state machine is in the idle/hold state and the microprocessor is accessing the content addressable memory(ies) (68KEN and 68KACC will also be asserted by the state decode). In this case all packets are to be discarded and XFAIL will be asserted. If NOPASS is asserted, then the state machine has been placed in a mode where is only learning addresses on the local ring. This is controlled by a signal to the state machine from the microprocessor called LEARN ONLY. If this signal is asserted, then NOPASS will be asserted, which will cause XFAIL to be asserted. Then the state machine will advance to Wait For End of Packet 254, and assert the packet counter signal, PCKCNT, that allows the microprocessor to keep track of the number of packets, whether they are filtered or not.

This state machine is the same for the Ethernet case, except that the signals have somewhat different names, corresponding to the way they are produced. XMATCH and XFAIL as inputs to the state machine are combined into a single signal called RENA (for Receive ENAble). As outputs from the external decoder 503 XMATCH and XFAIL are also combined to a single signal, called CLSN EN (for CoLiSioN ENable). The ADDMATCH0 and ADDMATCH1 signals are combined into COUNT 16 signal, which indicates that the next 16-bits of a destination or source address are available. There is no TSAMATCH signal since in Ethernet the transmitter does not need to strip its transmitted packet.

With respect to the state decode 96, only the signals PE and CEP are not used. All other signals operate in the same manner as for the Token Ring.

The system of the present invention has been described above as being applicable to the interfacing of LANs of the Ethernet type and of the Token Ring type. The system could also be used with the interfacing of other types of LANs, such as the types described in detail in the publication *Understanding Local Area Networks* referred to above.

Therefore, while the foregoing invention has been described with reference to its preferred embodiment, various alterations and modifications will occur to those skilled in the art. All such modifications and alterations are intended to fall within the scope of the appended claims.

```
Name           lafsU1e;
Partno      27-260-3;
Assemoly    LAFS;
Date        06/23/89;
Revision    2.1:
Designer    g.b. videlock/d. nadrowski;
Company     microcom, inc;
Location    U1;
Device      p22v10;

/*******************************************************************/
/*  Allowable Target Device Types:  22v10-15                     */
/*******************************************************************/

/**  Inputs  **/

Pin 1         =  lclk        ;
Pin 2         =  !xmatch     ;
Pin 3         =  !xfail      ;
Pin 4         =  learnonly   ;
PIN 5         =  !addmatch0  ;
Pin 6         =  !addmatch1  ;
Pin 7         =  !hndshk     ;
Pin 8         =  !allfull    ;
Pin 9         =  !tsamatch   ;
Pin 10        =  !matchany   ;
Pin 11        =   slvhold    ;   /* active high to match spec */
Pin 13        =  !mac        ;

/**  Outputs  **/

Pin 23       = !q1        ;   /* STA */
Pin 22       = !q6        ;   /* STB */
Pin 21       = !q9        ;   /* STC */
Pin 20       = !q2        ;   /* STD */
Pin 19       = !q0        ;   /* STE */
Pin 18       = !q5        ;   /* STF */
Pin 17       = !q4        ;   /* STG */
Pin 16       = !q3        ;   /* STH */
Pin 15       = !q8        ;   /* STJ */
Pin 14       = !q7        ;   /* STK */

/** Declarations and Intermediate Variable Definitions **/

$define and      &
$define or       #
$define ON       'b'1
$define off      'b'0
```

```
FIELD st = [q9..0];

$define s0        'b'0000000000
$define s1        'b'0000000001
$define s1a       'b'0000010001   /*rst seg cnt 'BOXX'*/
$define s1b       'b'0000010101
$define s2        'b'1000000000   /* addmatch */
$define s3        'b'1000000010
$define s4        'b'1000000011
$define s5        'b'1001000000   /* addmatch */
$define s6        'b'1000000100
$define s7        'b'1000000101
$define s8        'b'1010000000   /* addmatch */
$define s9        'b'1000001000
$define s10       'b'1000001100
$define s11       'b'1000001010
$define s12       'b'1011000000   /* addmatch */
$define s13       'b'1000010000
$define s14       'b'10[illegible]10001
$define s15       'b'11[illegible]00000   /* addmatch */
$define s16       'b'1000010100
$define s17       'b'1000011100
$define s18       'b'1101000000   /* addmatch */
$define s19       'b'1000100000
$define s20       'b'1000000001
$define s21       'b'0000000011
$define s22       'b'0000100010
$define s23       'b'0000100110
$define s24       'b'0000100111
$define s25       'b'0000100101
$define s25a      'b'0000100000
$define s25b      'b'0000100001
$define s25c      'b'0000100100
$define s26       'b'1000100100
$define s27       'b'1110000000   /* addmatch */
$define s28       'b'1000101000
$define s29       'b'1000101100
$define s30       'b'1111000000   /* addmatch */
$define s31       'b'1000101010
$define s32       'b'1000000111
$define s33       'b'0000101000
$define s34       'b'0000001000
$define s35       'b'0000001100
$define s36       'b'0000110000
$define s37       'b'0000110001
$define s38       'b'0010000000  /* new addmatch */
$define s39       'b'0000110100
$define s40       'b'0001000000  /* new addmatch */

/**  Logic Equations  **/

q0.ar  = learnonly and slvhold;
q1.ar  = learnonly and slvhold;
q2.ar  = learnonly and slvhold;
q3.ar  = learnonly and slvhold;
q4.ar  = learnonly and slvhold;
q5.ar  = learnonly and slvhold;
q6.ar  = learnonly and slvhold;
q7.ar  = learnonly and slvhold;
q8.ar  = learnonly and slvhold;
q9.ar  = learnonly and slvhold;
q0.sp  = off;
q1.sp  = off;
q2.sp  = off;
q3.sp  = off;
q4.sp  = off;
q5.sp  = off;
q6.sp  = off;
q7.sp  = off;
q8.sp  = off;
q9.sp  = off;

sequence st {
```

```
present s0
        if xmatch & xfail                           next s1;
        if slvhold                                  next s36;
        default                                     next s0;

present s1
        if mac                                      next s40;
        default                                     next s1a;

present s1a
        default                                     next s1b;

present s1b
        default                                     next s2;

present s2
        if !addmatch0 or !addmatch1                 next s2;
        default                                     next s3;

present s3
                                                    next s4;

present s4
                                                    next s5;

present s5
        if !addmatch0 or !addmatch1                 next s5;
        default                                     next s6;

present s6
                                                    next s7;

present s7
                                                    next s8;

present s8
        if !addmatch0 or !addmatch1                 next s8;
        default                                     next s9;

present s9
                                                    next s10;

present s10
                                                    next s11;

present s11
        if allfull or hndshk                        next s33;
        default                                     next s12;

present s12
        if !addmatch0 or !addmatch1                 next s12;
        default                                     next s13;

present s13
        if tsamatch                                 next s26;
        default                                     next s14;

present s14
                                                    next s15;

present s15
        if !addmatch0 or !addmatch1                 next s15;
        default                                     next s16;

present s16
                                                    next s17;

present s17
                                                    next s18;

present s18
        if !addmatch0 or !addmatch1                 next s18;
        default                                     next s19;

present s19
                                                    next s20;
```

```
present s20
                                                        next s21;

present s21
        if matchany                                     next s33;
        default                                         next s22;

present s22
                                                        next s23;

present s23
                                                        next s24;

present s24
                                                        next s25;

present s25
                                                        next s25a;

present s25a
                                                        next s25b;

present s25b
                                                        next s25c;

present s25c
                                                        next s33;

present s26
                                                        next s27;

present s27
        if !addmatch0 or !addmatch1                     next s27;
        default                                         next s28;

present s28
        if !tsamatch                                    next s17;
        default                                         next s29;

present s29
                                                        next s30;

present s30
        if !addmatch0 or !addmatch1                     next s30;
        default                                         next s31;

present s31
        if !tsamatch                                    next s20;
        default                                         next s32;

present s32
                                                        next s35;

present s33
        if learnonly                                    next s35;
        default                                         next s34;

present s34
                                                        next s0;
present s35
                                                        next s0;

present s36
        if !slvhold and (!xmatch or !xfail)             next s0;
        if  xmatch and xfail                            next s37;
        default                                         next s36;

present s37
        default                                         next s38;

present s38
        if !addmatch0 or !addmatch1                     next s38;
        default                                         next s39;

present s39
                                                        next s36;
```

```
present s40
       if !addmatch0 or !addmatch1                           next s40;
       default                                               next s35;
}

/*******************************************************************/
/*  Allowable Target Device Types:  20L10-25                       */
/*******************************************************************/

/**  Inputs  **/

Pin 1        =   lclk      ;
Pin 2        =  !q1        ; /* STA */
Pin 3        =  !q6        ; /* STB */
Pin 4        =  !q9        ; /* STC */
Pin 5        =  !q2        ; /* STD */
Pin 6        =  !q0        ; /* STE */
Pin 7        =  !q5        ; /* STF */
Pin 8        =  !q4        ; /* STG */
Pin 9        =  !q3        ; /* STH */
Pin 10       =  !q8        ; /* STJ */
Pin 11       =  !q7        ; /* STK */

/**  Outputs  **/

Pin 14       = !pckcnt    ;
Pin 15       = !pe        ;
Pin 16       =  cmdcamlt  ;
Pin 17       = !cmdcam    ;
Pin 18       = !passcntrl ;
Pin 19       = !matchlatch;
Pin 20       = !strip     ;
Pin 21       = !rdx       ;
Pin 22       = !wrx       ;
Pin 23       = !csx       ;

/** Declarations and Intermediate Variable Definitions **/

$define and     &
$define or      #
$define ON      'b'1
$define off     'b'0

FIELD st = [q9..0];

$define s0      'b'0000000000
$define s1      'b'0000000001
$define s1a     'b'0000010001   /* rst seg cnt */
$define s1b     'b'0000010101
$define s2      'b'1000000000   /* addmatch */
$define s3      'b'1000000010
$define s4      'b'1000000011
$define s5      'b'1001000000   /* addmatch */
$define s6      'b'1000000100
$define s7      'b'1000000101
$define s8      'b'1010000000   /* addmatch */
$define s9      'b'1000001000
$define s10     'b'1000001100
$define s11     'b'1000001010
$define s12     'b'1011000000   /* addmatch */
$define s13     'b'1000010000
$define s14     'b'1000010001
$define s15     'b'1100000000   /* addmatch */
$define s16     'b'1000010100
$define s17     'b'1000011100
$define s18     'b'1101000000   /* addmatch */
$define s19     'b'1000100000
$define s20     'b'1000000001
$define s21     'b'0000100011
$define s22     'b'0000100010
$define s23     'b'0000100110
$define s24     'b'0000100111
$define s25     'b'0000100101
$define s25a    'b'0000100000
$define s25b    'b'0000100001
```

```
$define s25c    'b'0000100100
$define s26     'b'1000100100
$define s27     'b'1110000000   /* addmatch */
$define s28     'b'1000101000
$define s29     'b'1000101100
$define s30     'b'1111000000   /* addmatch */
$define s31     'b'1000101010
$define s32     'b'1000000111
$define s33     'b'0000101000
$define s34     'b'0000001000
$define s35     'b'0000001100
$define s36     'b'0000110000
$define s37     'b'0000110001
$define s38     'b'0010000000  /* new addmatch */
$define s39     'b'0000110100
$define s40     'b'0001000000  /* new addmatch */

/**   Logic Equations    **/

pckcnt = st:s34 # st:s35 # st:s39;

pe = st:s0 # st:s36;

!cmdcamlt = st:s22;

cmdcam = !q9 & !q8 & !q7 & !q6 & !q5 & !q3 & !q1 & q0/*st:s1 # st:s1a # st:s1b*/
     # !q9 & !q8 & !q7 & !q6 & q5 & !q4 & !q3 & q2/*st:s23#st:s24#st:s25#s25c */
     # st:s40:

passcntrl = st:s34;

matchlatch = st:s11:

strip = st:s39;

rdx = st:s22:

wrx = st:s1 # st:s1a # st:s24 # st:s25 # st:s40;

csx = st:s1 # st:s1a # st:s1b
    # !q9 & !q8 & !q7 & !q6 & q5 & !q4;/*s22#s23#s24#s25#s25a#s25b#s25c#
                                          s33#s40*/

/********************************************************************/
/*  Allowable Target Device Types:  20L8-15                          */
/********************************************************************/

/**  Inputs   **/

Pin 1        =   lclk      ;
Pin 2        =  !q1        ;     /* STA */
Pin 3        =  !q6        ;     /* STB */
Pin 4        =  !q9        ;     /* STC */
Pin 5        =  !q2        ;     /* STD */
Pin 6        =  !q0        ;     /* STE */
Pin 7        =  !q5        ;     /* STF */
Pin 8        =  !q4        ;     /* STG */
Pin 9        =  !q3        ;     /* STH */
Pin 10       =  !q8        ;     /* STJ */
Pin 11       =  !q7        ;     /* STK */
Pin 14       =  lbclk1     ;

/**  Outputs   **/

Pin 15       = !68ken     ;
Pin 16       = !encmux    ;
Pin 17       = !tmsltch   ;
Pin 18       = !salwrb    ;
Pin 19       = !salwra    ;
Pin 20       = !salwren   ;
Pin 21       = !wrall     ;
Pin 22       = !csall     ;


/** Declarations and Intermediate Variable Definitions **/
```

```
$define and     &
$define or      #
$define ON      'b'1
$define OFF     'b'0

FIELD st = [q9..0];

$define s0      'b'0000000000
$define s1      'b'0000000001
$define s1a     'b'0000010001   /* rst seg cnt */
$define s1b     'b'0000010101
$define s2      'b'1000000000   /* addmatch */
$define s3      'b'1000000010
$define s4      'b'1000000011
$define s5      'b'1001000000   /* addmatch */
$define s6      'b'1000000100
$define s7      'b'1000000101
$define s8      'b'1010000000   /* addmatch */
$define s9      'b'1000001000
$define s10     'b'1000001100
$define s11     'b'1000001010
$define s12     'b'1011000000   /* addmatch */
$define s13     'b'1000010000
$define s14     'b'1000010001
$define s15     'b'1100000000   /* addmatch */
$define s16     'b'1000010100
$define s17     'b'1000011100
$define s18     'b'1101000000   /* addmatch */
$define s19     'b'1000100000
$define s20     'b'1000000001
$define s21     'b'0000000011
$define s22     'b'0000100010
$define s23     'b'0000100110
$define s24     'b'0000100111
$define s25     'b'0000100101
$define s25a    'b'0000100000
$define s25b    'b'0000100001
$define s25c    'b'0000100100
$define s26     'b'1000100100
$define s27     'b'1110000000   /* addmatch */
$define s28     'b'1000101000
$define s29     'b'1000101100
$define s30     'b'1111000000   /* addmatch */
$define s31     'b'1000101010
$define s32     'b'1000000111
$define s33     'b'0000101000
$define s34     'b'0000001000
$define s35     'b'0000001100
$define s36     'b'0000110000
$define s37     'b'0000110001
$define s38     'b'0010000000  /* new addmatch */
$define s39     'b'0000110100
$define s40     'b'0001000000  /* new addmatch */

/**   Logic Equations   **/

68ken = st:s36 # st:s37 # st:s38 # st:s39;

encmux = st:s23;

tmsltch =  lbclk1;

salwrb =   st:s16 # st:s17 # st:s19 # st:s20 # st:s28 # st:s29 # st:s31
        #  st:s32;

salwra =   st:s16 # st:s17 #st:s22 # st:s23 # st:s24 # st:s28 # st:s29;

salwren = !lclk &
                (st:s14
               # st:s17
               # st:s20
               # st:s22
               # st:s26
               # st:s29);

wrall =        st:s1
             # st:s1a
             # st:s4
```

```
                # st:s7
                # st:s10
                # st:s14
                # st:s17
                # st:s20
                # st:s26
                # st:s29
                # st:s32
                # st:s37
                # st:s40;

csall.=  o9 # st:s1 # st:s1a # st:s1b # st:s21 # st:s40:

/*****************************************************************/
/*  Allowable Target Device Types:  20L8-15                     */
/*****************************************************************/

/**  Inputs  **/

Pin 1         =   lclk       ;
Pin 2         =  !q1         ;     /* STA */
Pin 3         =  !q6         ;     /* STB */
Pin 4         =  !q9         ;     /* STC */
Pin 5         =  !q2         ;     /* STD */
Pin 6         =  !q0         ;     /* STE */
Pin 7         =  !q5         ;     /* STF */
Pin 8         =  !q4         ;     /* STG */
Pin 9         =  !q3         ;     /* STH */
Pin 10        =  !q8         ;     /* STJ */
Pin 11        =  !q7         ;     /* STK */
Pin 14        =  !slvhold    ;

/**  Outputs  **/

Pin 15        = !nopass      ;
Pin 16        = !68kacc      ;
Pin 17        =  cep_inv     ;
Pin 18        = !cep         ;
Pin 19        = !clrtsa      ;
Pin 20        = !cmdcd12     ;
Pin 21        = !cmdcd14     ;
Pin 22        = !cdbufen     ;

/** Declarations and Intermediate Variable Definitions **/

$define and      &
$define or       #
$define ON       'b'1
$define OFF      'b'0

FIELD st = [q9..0];

$define s0       'b'0000000000
$define s1       'b'0000000001
$define s1a      'b'0000010001    /* rst seg cnt */
$define s1b      'b'0000010101
$define s2       'b'1000000000    /* addmatch */
$define s3       'b'1000000010
$define s4       'b'1000000011
$define s5       'b'1001000000    /* addmatch */
$define s6       'b'1000000100
$define s7       'b'1000000101
$define s8       'b'1010000000    /* addmatch */
$define s9       'b'1000001000
$define s10      'b'1000001100
$define s11      'b'1000001010
$define s12      'b'1011000000    /* addmatch */
$define s13      'b'1000010000
$define s14      'b'1000010001
$define s15      'b'1100000000    /* addmatch */
$define s16      'b'1000010100
$define s17      'b'1000011100
$define s18      'b'1100000000    /* addmatch */
$define s19      'b'1000100000
$define s20      'b'1000000001
```

```
$define s21       'b'0000100011
$define s22       'b'0000100010
$define s23       'b'0000100110
$define s24       'b'0000100111
$define s25       'b'0000100101
$define s25a      'b'0000100000
$define s25b      'b'0000100001
$define s25c      'b'0000100100
$define s26       'b'1000100100
$define s27       'b'1110000000   /* addmatch */
$define s28       'b'1000101000
$define s29       'b'1000101100
$define s30       'b'1111000000   /* addmatch */
$define s31       'b'1000101010
$define s32       'b'1000000111
$define s33       'b'0000101000
$define s34       'b'0000001000
$define s35       'b'0000001100
$define s36       'b'0000110000
$define s37       'b'0000110001
$define s38       'b'0010000000  /* new addmatch */
$define s39       'b'0000110100
$define s40       'b'0001000000  /* new addmatch */

/**   Logic Equations   **/

nopass = st:s35;

68kacc = st:s36 # st:s37 # st:s38 # st:s39;

cep =           -st:s1  # st:s4 # st:s7 # st:s10
               # st:s14 # st:s17 # st:s20 # st:s26
               # st:s29 # st:s32 # st:s37;


cep_inv = cep;

clrtsa = st:s32;

cmdcd12 = st:s23 #st:s24 # st:s25 # st:s25a;

cmdcd14 = st:s1 # st:s1a # st:s1b # st:s40;

cdbufen = q9;
```

What is claimed is:

1. An apparatus for transmitting information in packets between at least first and second data communication systems, each of said systems comprising a plurality of nodes, each of said nodes being adapted to receive and transmit packets of information, said packets of information each comprising destination address information, source address information and data, the apparatus comprising:

means for receiving a packet from the first of said data communications systems;

means for capturing destination address information from said received packet;

means for storing said captured destination address information;

means for comparing said captured destination address information to previously stored source addresses;

means f r capturing source address information from said received packet; and

means for storing said captured source address information;

wherein the comparison of said captured destination address information to previously stored source address information is completed before capturing said source address information from said received packet; and said packet received from said first data communications system is transmitted to said second data communications system only if said compared destination address information is not found by said means for comparing said captured destination address information to said previously stored source addresses.

2. The apparatus as claimed in claim 1, further comprising:

means for comparing said captured source address information to previously stored source addresses;

wherein said captured source address information will be added to said previously stored source addresses in said means for storing if the captured source address is not contained within said previously stored addresses.

3. A method of transmitting information in packets between at least first and second data communication systems, each of said systems comprising a plurality of nodes, each of said nodes being adapted to receive and transmit packets of information, said packets of information each comprising destination address information, source address information and data, said method comprising the steps of:

receiving a packet from the first of said data communications system;

capturing destination address information from said received packet;

storing said captured destination address information;

comparing said captured destination address information to previously stored source addresses;

capturing source address information from said received packet; and

storing said captured source address information;

wherein the comparison of said captured destination address information to previously stored source address information is completed before capturing said source address information from said received packet; and the transmission of said packet received from said first data communications system to said second data communications system is executed only if said compared destination address information is not found by the step of comparing said captured destination address information to said previously stored source addresses.

4. The method as claimed in claim 3, further comprising the steps of:

comparing said captured source address information to previously stored source addresses; and

adding said captured source address information to said previously stored source addresses if the captured source address is not contained within said previously stored addresses.

5. An apparatus for transmitting information in packets in a data communications system including first and second data communication subsystems, each of said subsystems comprising a plurality of nodes, each of said nodes being adapted to receive and transmit packets of information to and from the other of said nodes, said packets of information each comprising destination address information, source address information and data, the apparatus comprising:

means for receiving a packet said first subsystems

means for capturing destination address information from said received packet;

means for storing said captured destination address information;

means for comparing said captured destination address information to previously stored source addresses;

means for capturing source address information from said received packet; and

means for storing said captured source address information;

wherein the comparison of said captured destination address information to previously stored source address information is completed before capturing said source address information from said received packet; and said packet received from said first subsystem is transmitted to said second subsystem only if said compared destination address information is not found by said means for comparing said captured destination address information to said previously stored source addresses.

6. The apparatus as claimed in claim 5, wherein each of said subsystems are local area networks.

7. The apparatus as claimed in claim 6, wherein each of said local area networks complies substantially with IEEE standard 802.3 for Ethernet networks.

8. The apparatus as claimed in claim 6, wherein each of said local area networks complies substantially with IEEE standard 802.5 for Token Ring networks.

9. The apparatus as claimed in claim 6, wherein one of said local area networks complies substantially with one of IEEE standard 802.3 for Ethernet networks or IEEE standard 02.5 for Token Ring networks, and the other of said local area networks complies substantially with the other of said IEEE standard 802.3 for Ethernet networks or the IEEE standard 802.5 for Token Ring networks.

10. The apparatus as claimed in claim 5, said system further comprising:

means for comparing said captured source address information to previously stored source addresses; and

means for adding captured source address information to said previously stored source addresses if the captured source address is not contained within said previously stored addresses.

11. An attachment for a first data communications system adapted for receiving data packets from a second data communication system and for transmitting data packets to said second data communications system, each data communications system comprising a plurality of nodes, each of said nodes being adapted to receive and transmit packets of information, said packets of information each comprising destination address information, source address information and data, said attachment comprising:

interface means for receiving a packet from the first of said data communications system;

means for capturing destination address information from said received packet;

means for storing said captured destination address information;

means for comparing said captured destination address information to previously stored source addresses;

means for capturing source address information from said received packet; and

means for storing said captured source address information;

wherein the comparison of said captured destination address information to previously stored source address information is completed before capturing said source address information from said received packet; and said packet received from said first data communication system is transmitted to said second data communications system only if said compared destination address information is not found by said means for comparing said captured destination address information to said previously stored source addresses.

12. The attachment as claimed in claim 11, further including means for selectively interfacing with the data communications system through both parallel and serial data paths.

13. The attachment in claim 12, wherein said selectively interfacing means for interfacing selectively in-

terfaces with said data communications system (a) through said parallel data path when said first data communications network substantially complies with IEEE 802.3 network standard for Ethernet, and (b) through said series data path when said first data communications network substantially complies with IEEE 802.5 network standard for Token Ring.

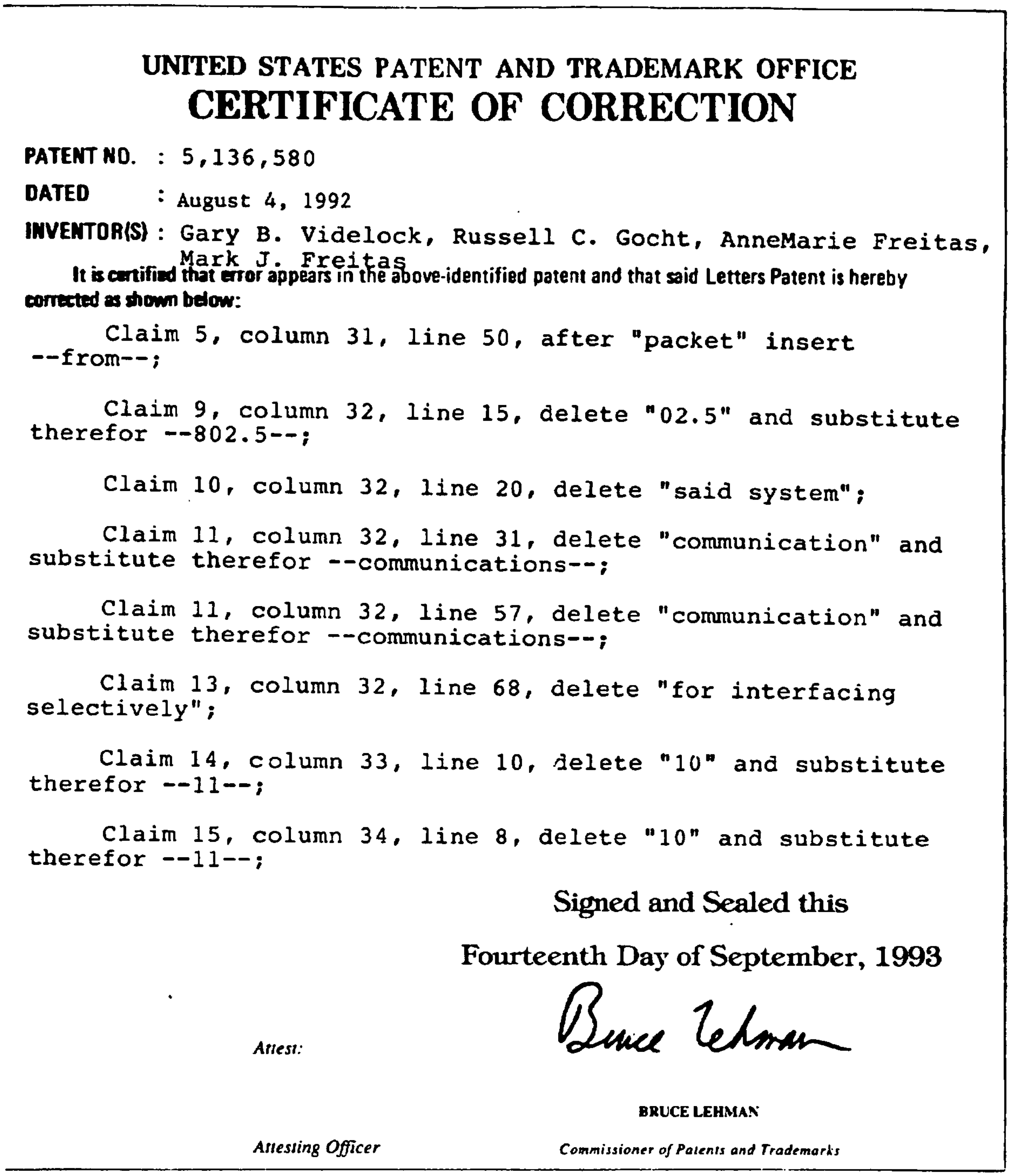
14. The attachment as claimed in claim 10, further comprising:

means on said attachment for comparing said captured source address information to previously stored source addresses;

wherein said captured source address information will be added to said previously stored source addresses if the captured source address is not contained within said previously stored addresses.

15. The attachment as claimed in claim 10, wherein said attachment to said first data communications system is made by way of a parallel data path.

* * * * *



UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,136,580

DATED : August 4, 1992

INVENTOR(S) : Gary B. Videlock, Russell C. Gocht, AnneMarie Freitas, Mark J. Freitas

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Claim 5, column 31, line 50, after "packet" insert --from--;

Claim 9, column 32, line 15, delete "02.5" and substitute therefor --802.5--;

Claim 10, column 32, line 20, delete "said system";

Claim 11, column 32, line 31, delete "communication" and substitute therefor --communications--;

Claim 11, column 32, line 57, delete "communication" and substitute therefor --communications--;

Claim 13, column 32, line 68, delete "for interfacing selectively";

Claim 14, column 33, line 10, delete "10" and substitute therefor --11--;

Claim 15, column 34, line 8, delete "10" and substitute therefor --11--;

Signed and Sealed this

Fourteenth Day of September, 1993

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US005696976A

United States Patent [19]

Nizar et al.

[11] **Patent Number: 5,696,976**

[45] **Date of Patent: Dec. 9, 1997**

[54] **PROTOCOL FOR INTERRUPT BUS ARBITRATION IN A MULTI-PROCESSOR SYSTEM**

[75] Inventors: **P. K. Nizar**, El Dorado Hills, Calif.; **David Carson**, Hillsboro, Oreg.

[73] Assignee: **Intel Corporation**, Santa Clara, Calif.

[21] Appl. No.: **710,452**

[22] Filed: **Sep. 17, 1996**

Related U.S. Application Data

[63] Continuation of Ser. No. 643,734, May 6, 1996, Pat. No. 5,613,128, which is a continuation of Ser. No. 49,515, Apr. 19, 1993, abandoned, which is a continuation-in-part of Ser. No. 8,074, Jan. 22, 1993, Pat. No. 5,283,904, which is a continuation of Ser. No. 632,149, Dec. 21, 1990, abandoned.

[51] **Int. Cl.**^{6} **G06F 13/26**

[52] **U.S. Cl.** **395/739**; 395/733; 395/737; 395/868

[58] **Field of Search** 395/733, 734, 395/736, 737, 739, 741, 738, 742, 868, 860, 728

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,812,463	5/1974	Lahti et al.	395/742
3,895,353	7/1975	Dalton	395/736
3,905,025	9/1975	Davis et al.	395/732
4,209,838	6/1980	Alcom, Jr. et al.	395/299
4,250,546	2/1981	Boney et al.	395/735
4,268,904	5/1981	Suzuki et al.	395/860
4,271,468	6/1981	Christensen et al.	395/859
4,394,730	7/1983	Suzuki et al.	395/650
4,402,040	8/1983	Evett	395/299
4,420,806	12/1983	Johnson, Jr. et al.	395/742
4,435,780	3/1984	Herrington et al.	395/732
4,482,954	11/1984	Vrielink et al.	395/741
4,484,264	11/1984	Friedli et al.	395/200.2
4,495,569	1/1985	Kagawa	364/200
4,621,342	11/1986	Capizzi et al.	395/291
4,648,029	3/1987	Cooper et al.	395/293
4,654,820	3/1987	Brahm et al.	395/306
4,788,639	11/1988	Tamaru	395/737
4,796,176	1/1989	D'Amico et al.	395/863
4,799,148	1/1989	Nishioka	395/738
4,805,096	2/1989	Crohn	395/733
4,833,598	5/1989	Imamura et al.	395/650
4,839,800	6/1989	Barlow et al.	395/737
4,860,196	8/1989	Wengert	395/550

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

1357576	6/1974	United Kingdom .

OTHER PUBLICATIONS

Examiner's Report to the Comptroller under Section 17 (The Search Report)—re: Application No. GB 9402811.5, completed May 11, 1994.

L.C. Eggebrecht, "Interfacing to the IBM Personal Computer" pp. 150–153 (1990).

Popescu, et al. "The Metaflow Architecture" pp. 10–73 IEEE Micro (Jun., 1991).

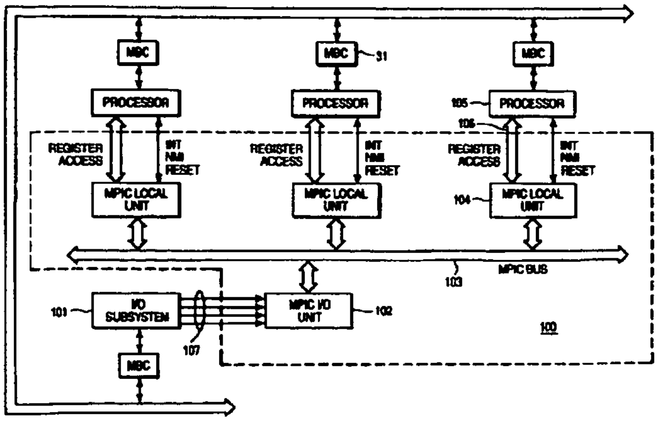
Thorne, M. "Computer Organization and Assembly Language Programming for IBM PC's and Compatibles", 2nd Ed., pp. 537–561.

Primary Examiner—Gopal C. Ray
Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman

[57] **ABSTRACT**

A multi-processor system includes an interrupt bus used for arbitrating among eligible processors to determine which processor is to service of an interrupt request. The interrupt bus comprises wired-OR connection data lines that are used for arbitration. A local interrupt controller that handles the acceptance of interrupt request messages on the interrupt bus is associated with each processor. To minimize interruption of high priority tasks, interrupts can be accepted by the processor in the system that is currently running the lowest priority task. An arbitration protocol governs the interrupt bus and determines the lowest priority processor. The arbitration protocol includes choosing one among the lowest priority processors by means of a random priority scheme that uses an arbitration ID that is updated with each message.

12 Claims, 12 Drawing Sheets



U.S. PATENT DOCUMENTS

4,866,664	9/1989	Burkhardt, Jr. et al.	395/200.03
4,868,742	9/1989	Gant et al.	395/850
4,903,270	2/1990	Johnson et al.	371/68.1
4,914,580	4/1990	Jensen et al.	395/738
4,920,486	4/1990	Nielsen	395/291
4,930,070	5/1990	Yonekura et al.	395/735
4,953,072	8/1990	Williams	395/741
4,980,854	12/1990	Donaldson et al.	395/297
5,060,139	10/1991	Theus	395/303
5,067,071	11/1991	Schanin et al.	395/293
5,083,261	1/1992	Wilkie	395/738
5,099,414	3/1992	Cole et al.	395/200.06
5,101,497	3/1992	Culley et al.	395/734
5,123,094	6/1992	MacDougall	395/375
5,125,093	6/1992	McFarland	395/739
5,134,706	7/1992	Cushing et al.	395/725
5,146,597	9/1992	Williams	395/741
5,155,853	10/1992	Mitsuhira et al.	395/725
5,179,707	1/1993	Piepho	395/733
5,193,187	3/1993	Strout, II et al.	395/650
5,201,051	4/1993	Koide	395/741
5,210,828	5/1993	Bolan et al.	395/200.08
5,218,703	6/1993	Fleck et al.	395/737
5,261,107	11/1993	Klim et al.	395/739
5,265,215	11/1993	Fukuda et al.	395/303
5,274,767	12/1993	Maskovyak	395/836
5,276,690	1/1994	Lee et al.	371/3
5,282,272	1/1994	Guy et al.	395/200.06
5,283,869	2/1994	Adams et al.	395/200.2
5,283,904	2/1994	Carson et al.	395/739
5,325,536	6/1994	Chang et al.	395/736
5,410,710	4/1995	Sarangdhar et al.	395/739
5,428,794	6/1995	Williams	395/741

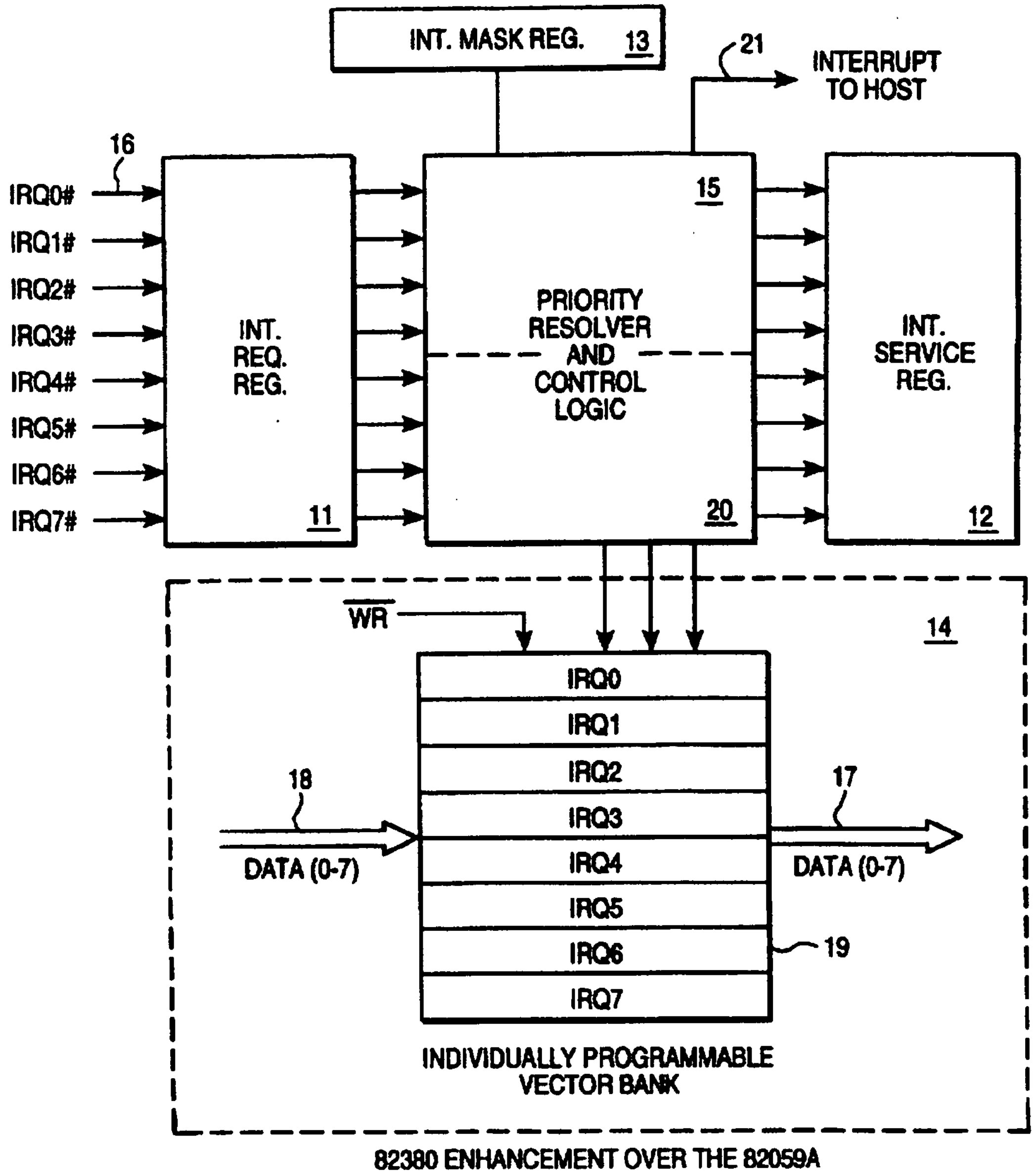


FIG_1 (PRIOR ART)

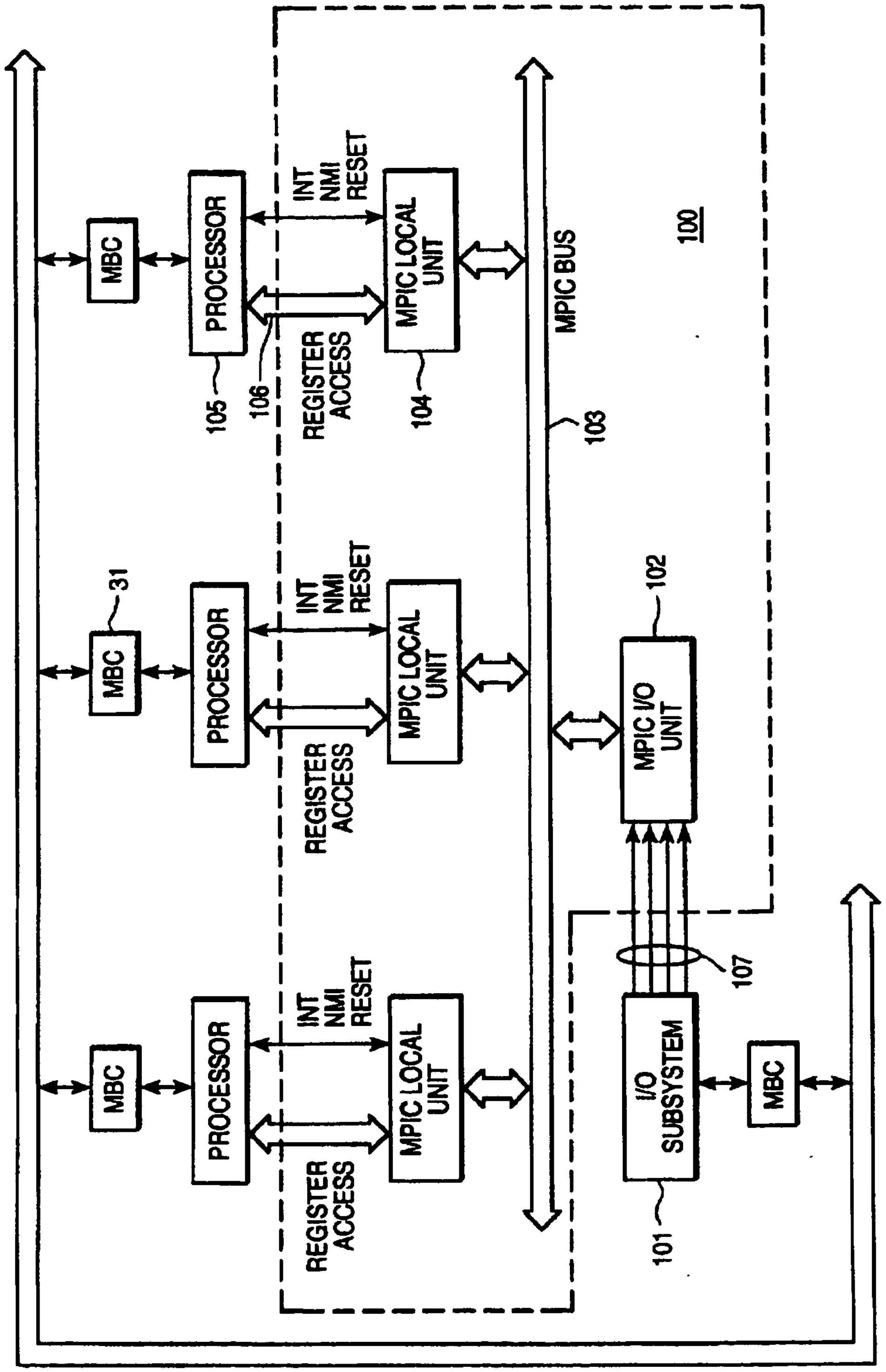

MBC
PROCESSOR
INT
NMI
RESET
REGISTER ACCESS
MPIC LOCAL UNIT
MPIC BUS
100
105
106
104
103
MBC
31
PROCESSOR
INT
NMI
RESET
REGISTER ACCESS
MPIC LOCAL UNIT
MPIC I/O UNIT
102
MBC
PROCESSOR
INT
NMI
RESET
REGISTER ACCESS
MPIC LOCAL UNIT
I/O SUBSYSTEM
101
107
MBC

FIG_2

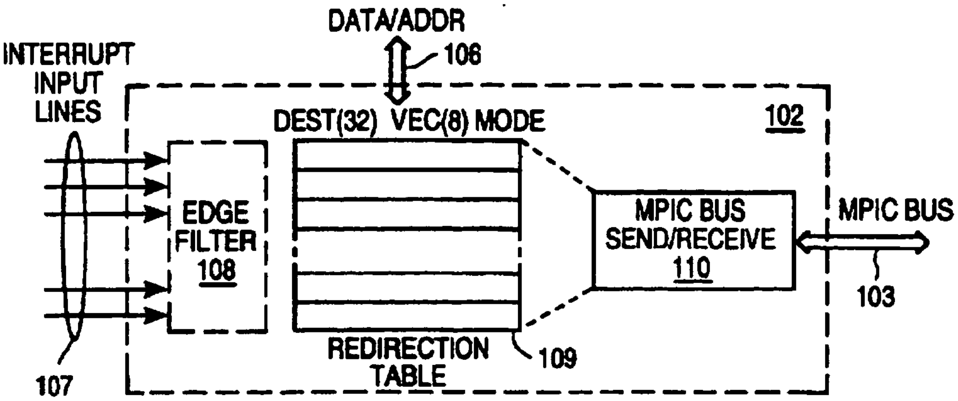

DATA/ADDR
106
INTERRUPT INPUT LINES
DEST(32) VEC(8) MODE
102
EDGE FILTER 108
MPIC BUS SEND/RECEIVE 110
MPIC BUS
103
REDIRECTION TABLE
109
107

FIG_3

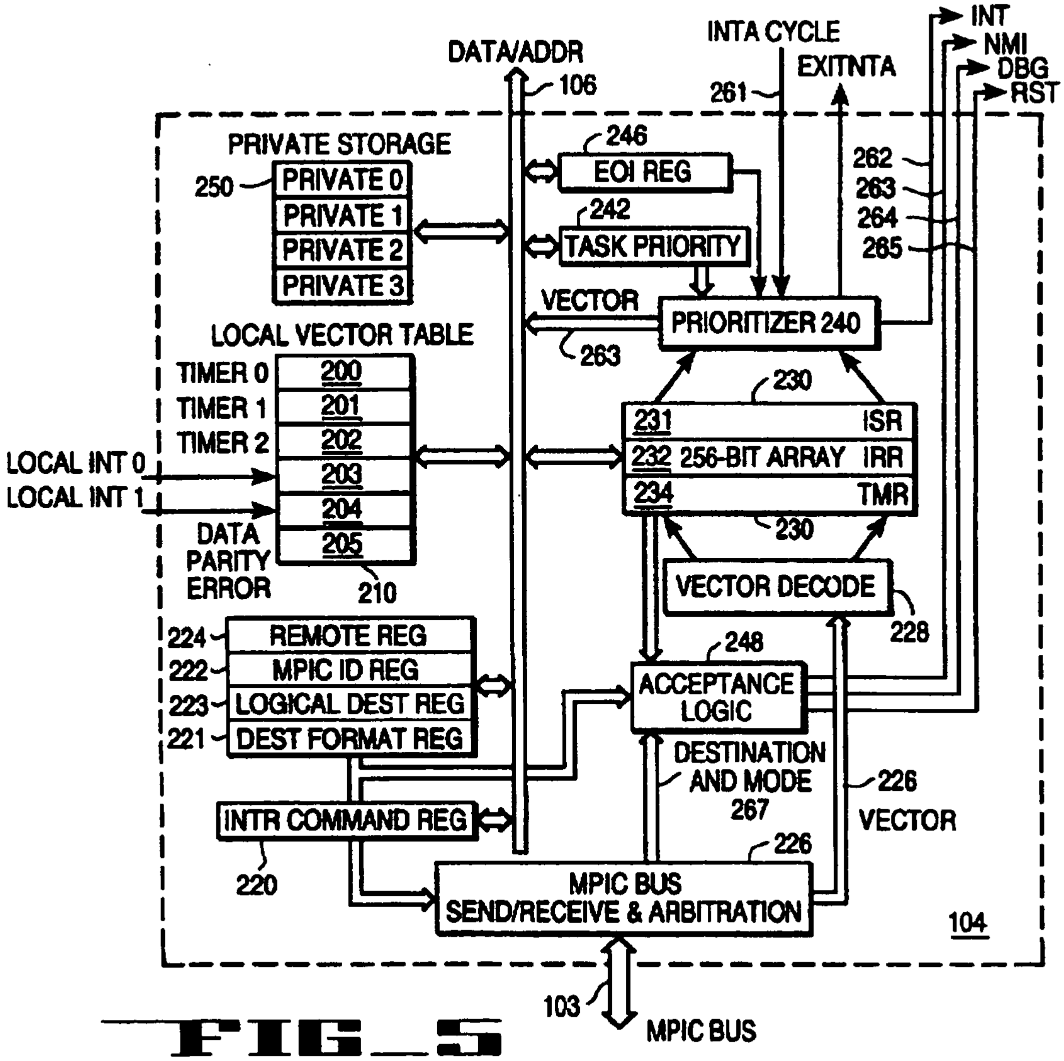

DATA/ADDR
106
INTA CYCLE
261
EXITNTA
INT
NMI
DBG
RST
PRIVATE STORAGE
250
PRIVATE 0
PRIVATE 1
PRIVATE 2
PRIVATE 3
246
EOI REG
242
TASK PRIORITY
262
263
264
265
VECTOR
PRIORITIZER 240
263
LOCAL VECTOR TABLE
TIMER 0
TIMER 1
TIMER 2
200
201
202
203
204
205
LOCAL INT 0
LOCAL INT 1
DATA PARITY ERROR
210
230
231 ISR
232 256-BIT ARRAY IRR
234 TMR
230
VECTOR DECODE
228
224
222
223
221
REMOTE REG
MPIC ID REG
LOGICAL DEST REG
DEST FORMAT REG
248
ACCEPTANCE LOGIC
DESTINATION AND MODE 267
226
VECTOR
INTR COMMAND REG
220
226
MPIC BUS SEND/RECEIVE & ARBITRATION
104
103
MPIC BUS

FIG_5

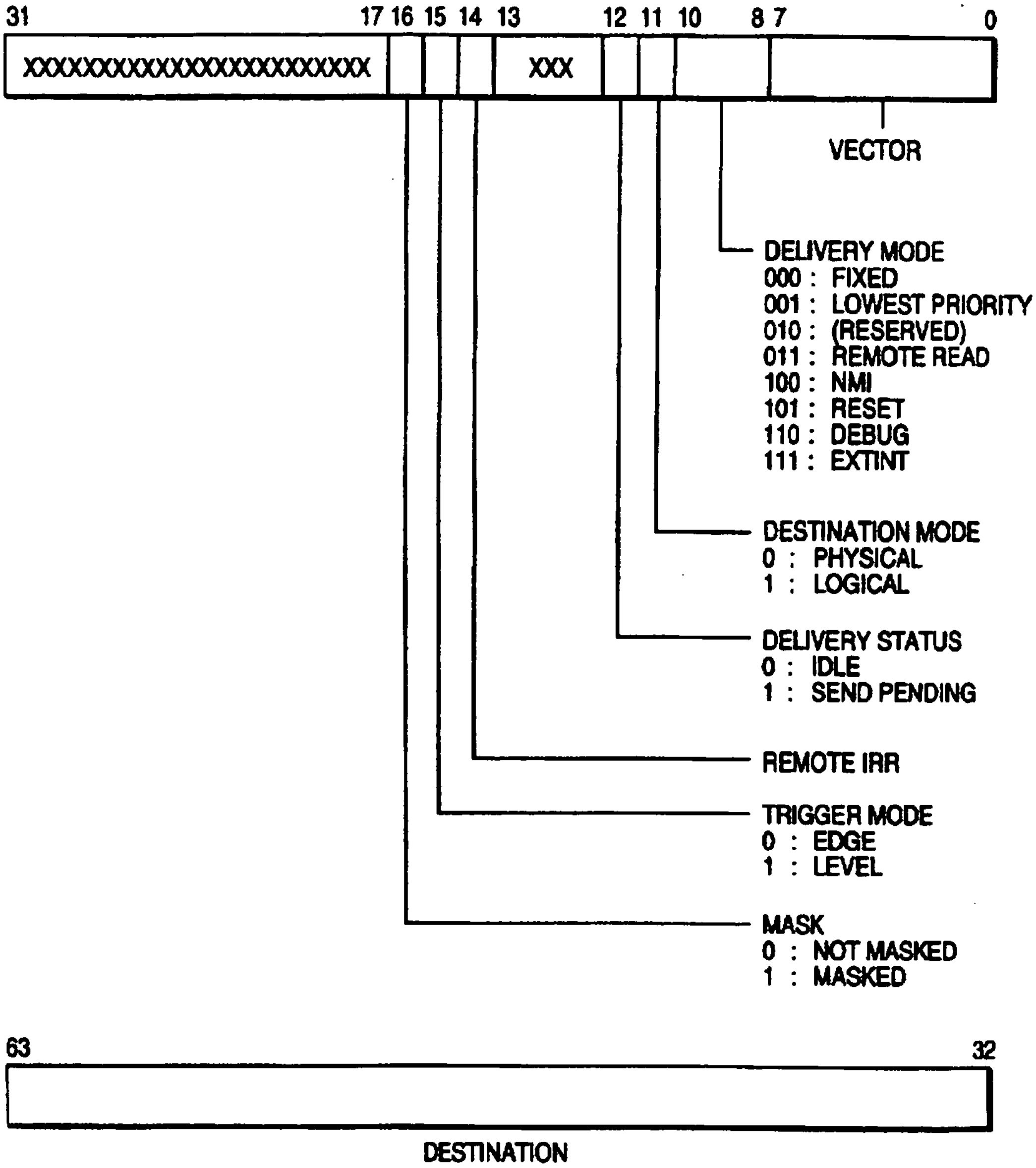

31
17 16 15 14 13
12 11 10
8 7
0
XXXXXXXXXXXXXXXXXXXXXXX
XXX
VECTOR
DELIVERY MODE
000 : FIXED
001 : LOWEST PRIORITY
010 : (RESERVED)
011 : REMOTE READ
100 : NMI
101 : RESET
110 : DEBUG
111 : EXTINT
DESTINATION MODE
0 : PHYSICAL
1 : LOGICAL
DELIVERY STATUS
0 : IDLE
1 : SEND PENDING
REMOTE IRR
TRIGGER MODE
0 : EDGE
1 : LEVEL
MASK
0 : NOT MASKED
1 : MASKED
63
32
DESTINATION

FIG_4

	31 20	19 18	17	16	15		11	10 8	7 0
(200) TIMER 0	RESERVED	BASE	M	MS	RESERVED			DELIV	VECTOR
(201) TIMER 1	RESERVED	BASE	M	MS	RESERVED			DELIV	VECTOR
(202) TIMER 3	RESERVED	BASE	M	MS	RESERVED			DELIV	VECTOR
(203) LOCAL 0	RESERVED			MS	TM	R	RESERVED	DELIV	VECTOR
(204) LOCAL 1	RESERVED			MS	TM	R	RESERVED	DELIV	VECTOR
(205) PARITY ERROR	RESERVED			MS	RESERVED			DELIV	VECTOR

FIG_6

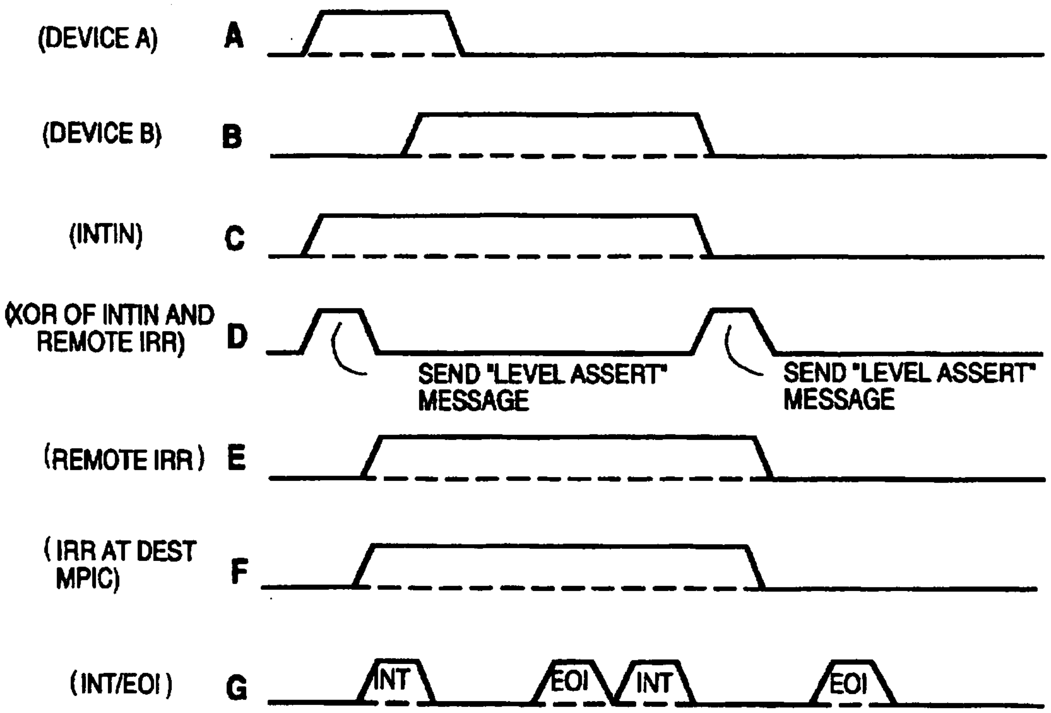


FIG_8

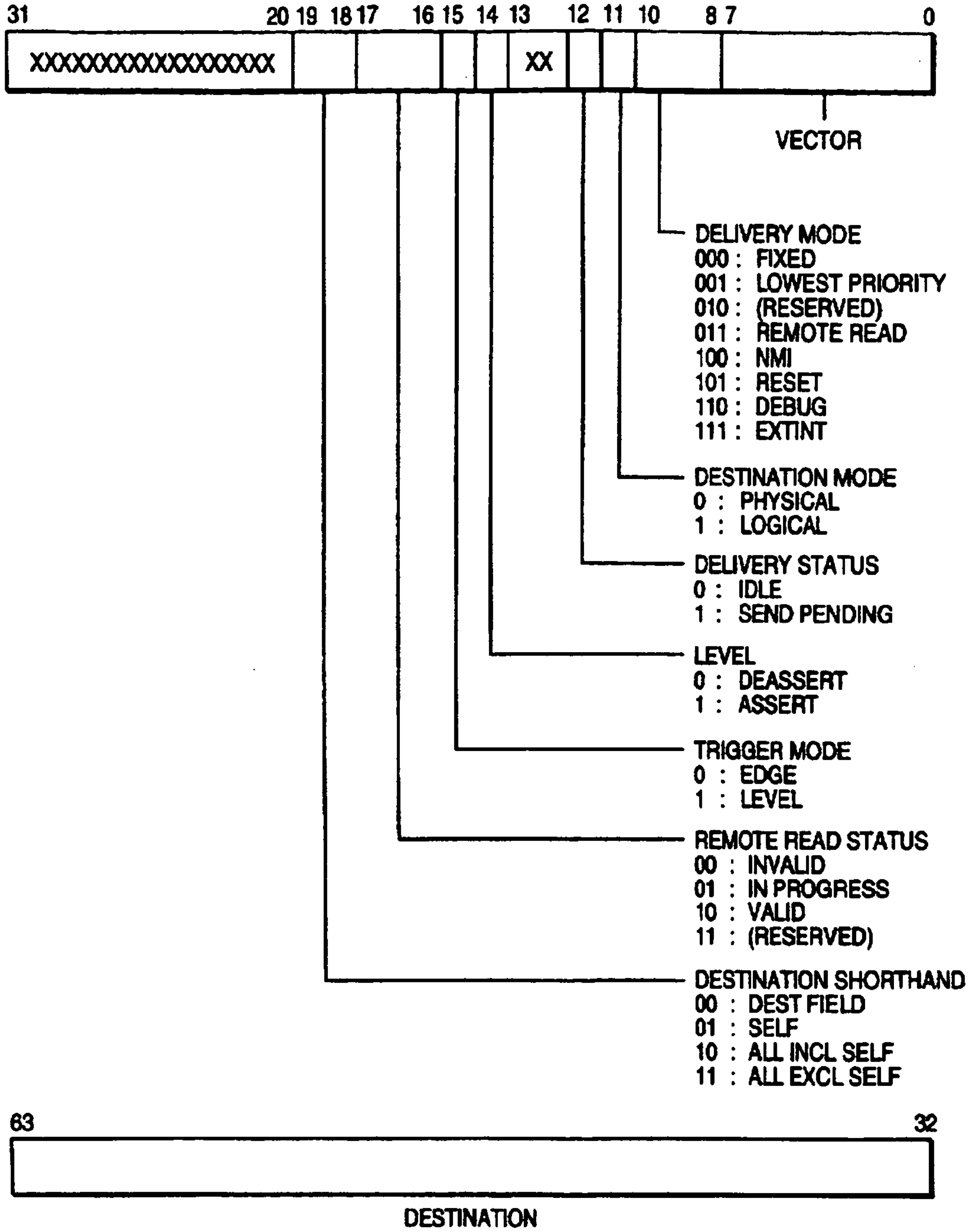

31
20 19 18 17 16 15 14 13 12 11 10 8 7 0
XXXXXXXXXXXXXXXXXX
XX
VECTOR
DELIVERY MODE
000 : FIXED
001 : LOWEST PRIORITY
010 : (RESERVED)
011 : REMOTE READ
100 : NMI
101 : RESET
110 : DEBUG
111 : EXTINT
DESTINATION MODE
0 : PHYSICAL
1 : LOGICAL
DELIVERY STATUS
0 : IDLE
1 : SEND PENDING
LEVEL
0 : DEASSERT
1 : ASSERT
TRIGGER MODE
0 : EDGE
1 : LEVEL
REMOTE READ STATUS
00 : INVALID
01 : IN PROGRESS
10 : VALID
11 : (RESERVED)
DESTINATION SHORTHAND
00 : DEST FIELD
01 : SELF
10 : ALL INCL SELF
11 : ALL EXCL SELF
63
32
DESTINATION

FIG_7

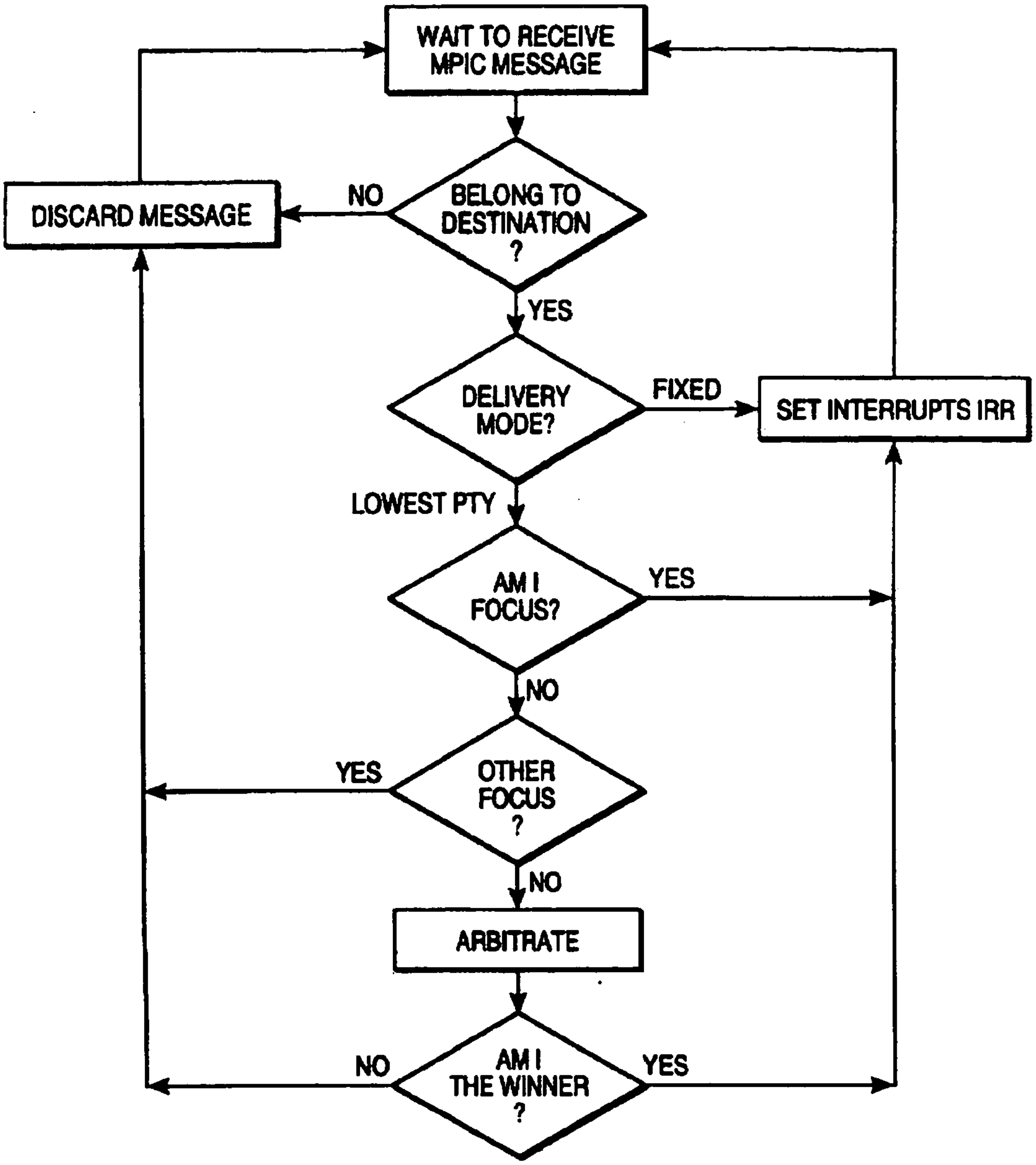

WAIT TO RECEIVE MPIC MESSAGE
BELONG TO DESTINATION ?
NO
DISCARD MESSAGE
YES
DELIVERY MODE?
FIXED
SET INTERRUPTS IRR
LOWEST PTY
AM I FOCUS?
YES
NO
OTHER FOCUS ?
YES
NO
ARBITRATE
AM I THE WINNER ?
NO
YES

FIG_9

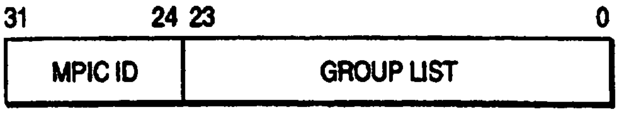

31
24 23
0
MPIC ID
GROUP LIST

FIG_10

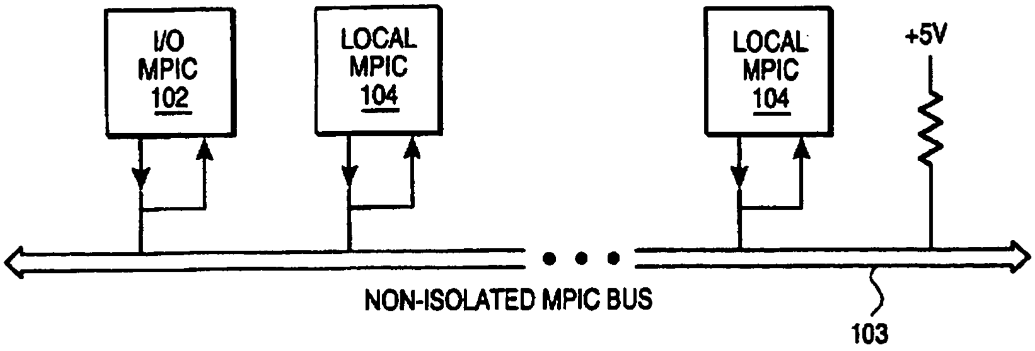

I/O MPIC 102
LOCAL MPIC 104
LOCAL MPIC 104
+5V
NON-ISOLATED MPIC BUS
103

FIG_11

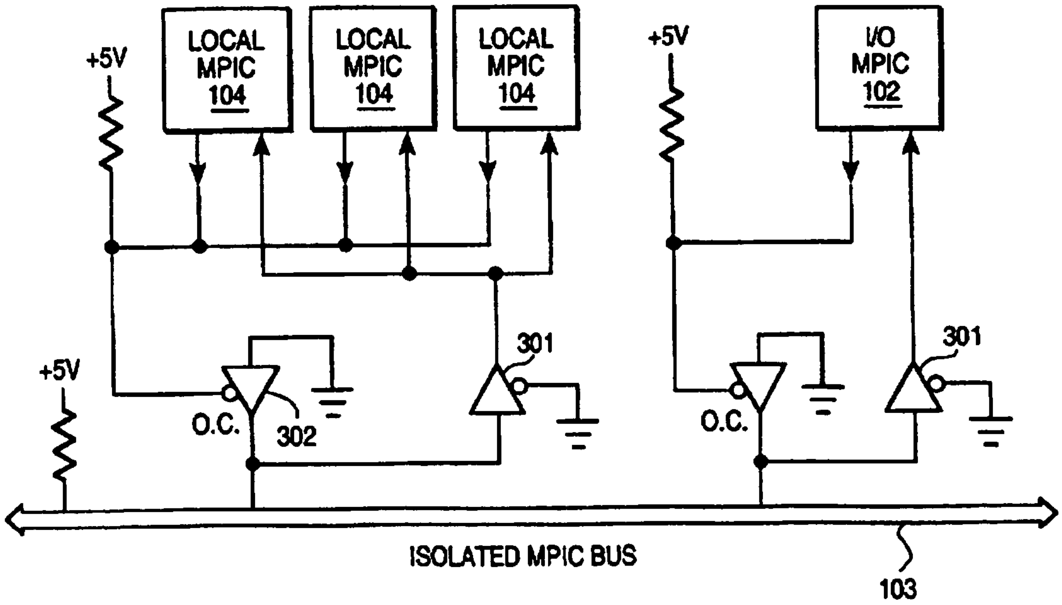

+5V
LOCAL MPIC 104
LOCAL MPIC 104
LOCAL MPIC 104
+5V
I/O MPIC 102
301
301
+5V
O.C.
302
O.C.
ISOLATED MPIC BUS
103

FIG_12

ID TUPLE			MPIC BUS			
(I [i+1]	I[i])	⟶	B3	B2	B1	B0
0	0	⟶	0	0	0	1
0	1	⟶	0	0	1	0
1	0	⟶	0	1	0	0
1	1	⟶	1	0	0	0

FIG_13

1 :	i76	i76	i76	i76	MPIC BUS ARBITRATION
2 :	i54	i54	i54	i54	
3 :	i32	i32	i32	i32	
4 :	i10	i10	i10	i10	
5 :	DM	M2	M1	M0	DESTINATION MODE AND DELIVERY MODE
6 :	"O"	"O"	L	TM	CONTROL BITS
7 :	V7	V6	V5	V4	VECTOR
8 :	V3	V2	V1	V0	
9 :	D31	D30	D29	D28	DESTINATION
10 :	D27	D26	D25	D24	
11 :	D23	D22	D211	D20	
12 :	D19	D18	D17	D16	
13 :	D15	D14	D13	D12	
14 :	D11	D10	D09	D08	
15 :	D07	D06	D05	D04	
16 :	D03	D02	D01	D00	
17 :	C	C	C	C	CHECKSUM FOR CYCLES 5 THROUGH 16
18 :	"1"	"1"	"1"	"1"	POSTAMBLE
19 :	A	A	A	A	ACCEPT (1000 IF OK, 1110 IF PREEEMPT, ELSE ERROR)
20 :	"O"	"O"	"O"	"O"	IDLE 1
21 :	"O"	"O"	"O"	"O"	IDLE 2

FIG_14

M2	M1	M0	DELIVERY MODE
0	0	0	FIXED
0	0	1	LOWEST PRIORITY
0	1	0	(RESERVED)
0	1	1	REMOTE READ
1	0	0	NMI
1	0	1	RESET
1	1	0	DEBUG
1	1	1	EXTINT

FIG_15

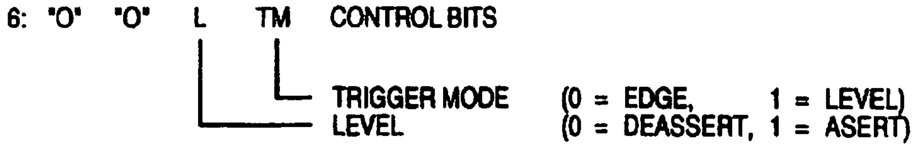

6: "O" "O" L TM CONTROL BITS
TRIGGER MODE (0 = EDGE, 1 = LEVEL)
LEVEL (0 = DEASSERT, 1 = ASERT)

FIG_16

TM/L (AAAA)

		EDGE	LEVEL-ASSERT	LEVEL-DEASSERT
DELEVERY MODE	FIXED	SHORT	SHORT	SHORT
	LOWEST PRIORITY	SHORT(1110)	SHORT(1110)	SHORT
		LONG(1000)	LONG(1000)	SHORT
	REMOTE READ	LONG	LONG	SHORT
	NMI	SHORT	SHORT	SHORT
	RESET	SHORT	SHORT	SHORT
	DEBUG	SHORT	SHORT	SHORT
	EXTINT	SHORT	SHORT	SHORT

FIG_17

1 :	¡76	¡76	¡76	¡76	MPIC BUS ARBITRATION
2 :	¡54	¡54	¡54	¡54	
3 :	¡32	¡32	¡32	¡32	
4 :	¡10	¡10	¡10	¡10	
5 :	DM	M2	M1	M0	DESTINATION MODE
6 :	"O"	"O"	L	TM	CONTROL BITS
7 :	V7	V6	V5	V4	VECTOR
8 :	V3	V2	V1	V0	
9 :	D31	D30	D29	D28	DESTINATION
10 :	D27	D26	D25	D24	
11 :	D23	D22	D211	D20	
12 :	D19	D18	D17	D16	
13 :	D15	D14	D13	D12	
14 :	D11	D10	D09	D08	
15 :	D07	D06	D05	D04	
16 :	D03	D02	D01	D00	
17 :	C	C	C	C	CHECKSUM FOR CYCLES 5 THROUGH 16
18 :	"1"	"1"	"1"	"1"	POSTAMBLE
19 :	A	A	A	A	ACCEPT (1000 IF OK, 1110 IF PREEEMPT, ELSE ERROR)
20 :	p76	p76	p76	p76	LOWEST PRIORITY ARBITRATION OR
21 :	p54	p54	p54	p54	32 BITS OF REMOTE REGISTRATION
22 :	p32	p32	p32	p32	
23 :	p10	p10	p10	p10	
24 :	a76	a76	a76	a76	
25 :	a54	a54	a54	a54	
26 :	a32	a32	a32	a32	
27 :	a10	a10	a10	a10	
28 :	A	A	A	A	ACCEPT
29 :	"O"	"O"	"O"	"O"	IDLE 1
30 :	"O"	"O"	"O"	"O"	IDLE 2

FIG_18

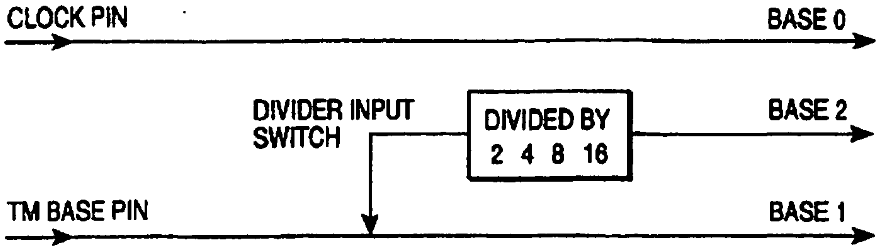


FIG_19

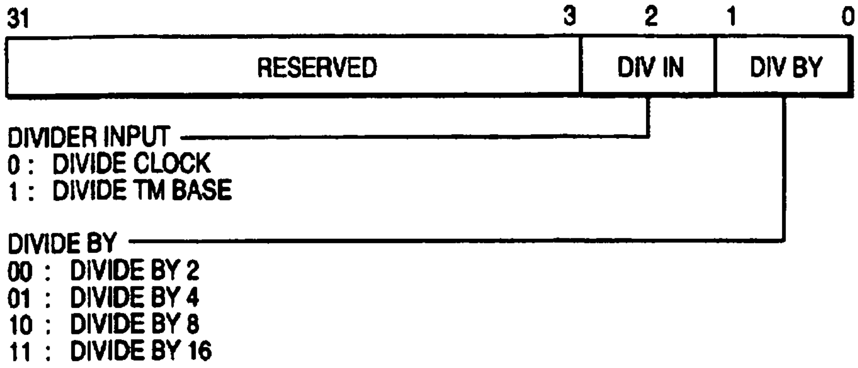


FIG_20

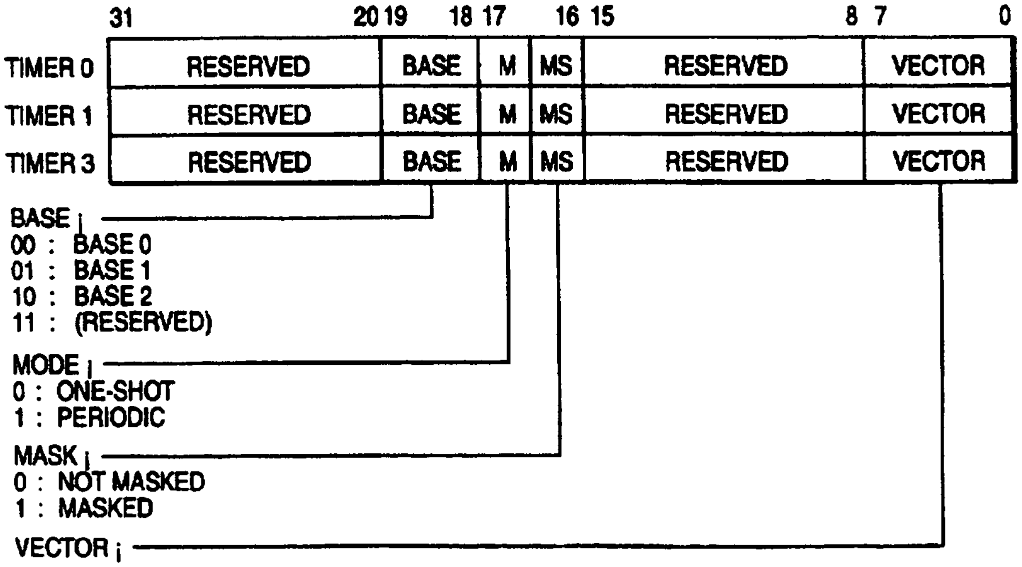


FIG_21

PROTOCOL FOR INTERRUPT BUS ARBITRATION IN A MULTI-PROCESSOR SYSTEM

CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation of Ser. No. 08/643,734, filed May 6, 1996, U.S. Pat. No. 5,613,128, which is a continuation of Ser. No. 08/049,515, filed Apr. 19, 1993, abandoned, which is a continuation in part of Ser. No. 08/008,074, filed Jan. 22, 1993, U.S. Pat. No. 5,283,904, which is a continuation of Ser. No. 07/632,149, filed Dec. 21, 1990, abandoned.

FIELD OF THE INVENTION

The present invention generally relates to the field of multi-processor systems and more specifically to interrupt controllers designed to manage peripheral equipment service interrupt requests in a multi-processor environment.

BACKGROUND OF THE INVENTION

Input/output peripheral equipment, including such computer items as printers, scanners and display devices require intermittent servicing by a host processor in order to ensure proper functioning. Services, for example, may include data delivery, data capture and/or control signals. Each peripheral will typically have a different servicing schedule that is not only dependent on the type of device but also on its programmed usage. The host processor is required to multiplex its servicing activity amongst these devices in accordance with their individual needs while running one or more background programs. Two methods for advising the host of a service need have been used: polled device and device interrupt methods. In the former method, each peripheral device is periodically checked to see if a flag has been set indicating a service request, while, in the latter method, the device service request is routed to an interrupt controller that can interrupt the host, forcing a branch from its current program to a special interrupt service routine. The interrupt method is advantageous because the host does not have to devote unnecessary clock cycles for polling. It is this latter method that the present invention addresses. The specific problem addressed by the current invention is the management of interrupts in a multi-processor system environment.

Multi-processor systems, often a set of networked computers having common peripheral devices, create a challenge in the design of interrupt control methods. For instance, in the case of a computer network servicing a number of users, it would be highly desirable to distribute the interrupt handling load in some optimum fashion. Processors that are processing high priority jobs should be relieved of this obligation when processors with lower priority jobs are available. Processors operating at the lowest priority should be uniformly burdened by the interrupt servicing requests. Also, special circumstances may require that a particular I/O device be serviced exclusively by a preselected (or focus) processor. Thus, the current invention addresses the problem of optimum dynamic and static interrupt servicing in multi-processor systems.

Prior art devices, exemplified by Intel's 82C59A and 82380 programmable interrupt controllers (PICs), are designed to accept a number of external interrupt request inputs. The essential structure of such controllers, shown in FIG. 1, consists of six major blocks:

IRR-	Interrupt Request Register 11 stores all interrupt levels (IRQx) on lines 16 requesting service;
ISR-	Interrupt Service Register 12 stores all interrupt levels which are being serviced, status being updated upon receipt of an end-of-interrupt (EOI);
IMR-	Interrupt Mask Register 13 stores the bits indicating which IRQ lines 16 are to be masked or disabled by operating on IRR11:
VR-	Vector Registers 19, a set of registers, one for each IRQ line 16, stores the pre-programmed interrupt vector number supplied to the host processor on data bus 17, containing all the necessary information for the host to service the request;
PR-	Priority Resolver 15, a logic block that determines the priority of the bits set in IRR11, the highest priority is selected and strobed into the corresponding bit of ISR12 during an interrupt acknowledge cycle (INTA) from the host processor;
Control Logic-	Coordinates the overall operations of the other internal blocks within the same PIC, activates the host input interrupt (INT) line 19 when one or more bits of IRR11 are active, enables VR19 to drive the interrupt vector onto data bus 17 during an INTA cycle, and inhibits all interrupts with priority equal or lower than that being currently serviced.

Several different methods have been used to assign priority to the various IRQ lines 16, including:

1) fully nested mode,

2) automatic rotation—equal priority devices, made and

3) specific rotation—specific priority mode.

The fully nested mode, supports a multi-level interrupt structure in which all of the IRQ input lines 16 are arranged from highest to lowest priority: typically IRQØ is assigned the highest priority, while IRQ7 is the lowest.

Automatic rotation of priorities when the interrupting devices am of equal priority is accomplished by rotating (circular shifting) the assigned priorities so that the most recently served IRQ line is assigned the lowest priority. In this way, accessibility to interrupt service tends to be statistically levelled for each of the competing devices.

The specific rotation method gives the user versatility by allowing the user to select which IRQ line is to receive the lowest priority, all other IRQ lines are then assigned sequentially (circularly) higher priorities.

From the foregoing description it may be seen that PIC structures of the type described accommodate uni-processor systems with multiple peripheral devices but do not accommodate multi-processor systems with multiple shared peripheral devices to which the present invention is addressed.

SUMMARY OF THE INVENTION

One object of the present invention is to provide for a multi-processor programmable interrupt controller (MPIC) system that uses an integrated circuit chip incorporating both the local processor and an mimed local processor interrupt controller as a single unit.

Another object is to provide a multi-processor programmable interrupt controller (MPIC) system including but not limited to the following capabilities:

1) multiple I/O peripheral devices, each with its own set of interrupts;

2)static as well as dynamic multi-processor interrupt management including the symmetrical distribution of interrupts over selected processors;

3) level or edge triggered interrupt request pins, software selectable per pin;

4) per pin programmable interrupt vector and steering information;

5) programmable vector address field defined by each operating system;

6) inter-processor interrupts allowing any processor to interrupt any other for dynamic reallocation of interrupt tasks; and

7) support of system wide support functions related to non-maskable interrupts (NMI), processor reset, and system debugging.

The present invention achieves these capabilities by means of an MPIC system structure which includes three major subsystems:

1) an I/O MPIC unit for acquiring interrupt request (IRQ) signals from its associated I/O peripheral devices, having a redirection table for processor selection and vector/priority information;

2) local-MPIC units which may be separate auxiliary units connected to the associated processor or units that am partially or totally integrated into the associated processor, each managing interrupt requests for a specific system processor including pending, nesting and masking operations, as well as inter-processor interrupt generation; and

3) a dedicated I/O bus, distinct from any system or memory bus, for communications between the I/O and local MPIC units as well as between local-MPIC units.

It is a further object of this invention to support system scaling granularity of one, i.e., one processor at a time without significant penalty for any number of processors.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be understood more fully from the detailed description given below and from the accompanying drawings of the preferred embodiments of the invention, which, however should not be taken to limit the invention to the specific embodiment but am for explanation and understanding only.

FIG. 1 depicts a block diagram of a common prior art uni-processor programmable interrupt controller (PIC).

FIG. 2 is a block diagram of the currently preferred multi-processor programmable interrupt controller (MPIC) system.

FIG. 3 is a block diagram of the currently preferred I/O-MPIC unit.

FIG. 4 shows the various fields that make-up a Redirection Table 64-bit entry.

FIG. 5 is a block diagram of the currently preferred local-MPIC unit.

FIG. 6 shows the various fields that constitute the local vector table entries of a local-MPIC unit.

FIG. 7 shows the various field assignments of the interrupt Command Register.

FIG. 8 depicts the tracking of the remote IRR bit by the destination IRR bit.

FIG. 9 is a flow chart depicting the interrupt acceptance process by a local-MPIC unit.

FIG. 10 shows the MPIC-ID register configuration.

FIG. 11 shows the non-isolated MPIC-Bus connections.

FIG. 12 shows a tri-state buffered MPIC-Bus arrangement.

FIG. 13 shows the 2-bit decode process for the MPIC-ID used in bus arbitration.

FIG. 14 shows the MPIC short message forms.

FIG. 15 shows the MPIC message encode of the delivery mode.

FIG. 16 defines the control bits of the MPIC message.

FIG. 17 defines the extended delivery mode control bit coding.

FIG. 18 shows the MPIC-Bus medium and long message formats.

FIG. 19 shows the Base 0, 1, and 2 time generator.

FIG. 20 shows the Divide (Base 2) Configuration Register bit assignments.

FIG. 21 shows the contents of the three timer Local Vector Table.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

A multi-processor programmable interrupt controller (MPIC), system is described. In the following description, numerous specific details are set forth, such as a specific number of input pins, bits, devices, etc.. in order to provide a thorough understanding of the preferred embodiment of the present invention. It will be obvious, however, to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known circuits have not been shown in detail, or have been shown in block diagram form only, in order to avoid unnecessarily obscuring the present invention.

Additionally, in describing the present invention, reference is made to signal names peculiar to the currently preferred embodiment. Reference to these specific names should not be construed as a limitation on the spirit or scope of the present invention.

A. Overview of the MPIC Architecture

The multi-processor programmable interrupt controller (MPIC) system is designed to accommodate interrupt servicing in a multi-processor environment. Current practice is mainly concerned with uni-processor systems in which the interrupts of a number of peripheral units are serviced by a single processor aided by a programmable interrupt controller (PIC). In a multi-processor system, it is often desirable to share the burden of interrupt servicing among a group of similar processors. This implies the ability to broadcast interrupt service requests to the pertinent group of processors and a mechanism for determining the equitable assignment of the tasks amongst the processors. The uni-processor design problem is significantly simpler: the PIC dedicated to the processor assigns a priority to each interrupt request (IRQ) line, orders the requests according to the assigned priorities and delivers the necessary information to the processor to timely initiate the appropriate servicing subroutine.

The MPIC system provides both static and dynamic interrupt task assignment to the various processors. When operating in a purely static mode, it functions much as a PIC in a uni-processor system assigning each interrupt according to a prescribed schedule.

When operating in a dynamic mode, the MPIC manages interrupt task assignment by taking into consideration the relative task priority between processors.

It is expected that more typical usage would entail elements of both static and dynamic interrupt management. Static assignments might be made, for example, when licensing considerations preclude the shared use of servicing software. Under other circumstances it may be desirable to restrict the interrupt servicing task to a subset of processors that share a common peripheral subsystem. In the extreme case, all processors are subject to interrupt requests from all peripheral subsystems.

FIG. 2 is a block diagram of the currently preferred multi-processor programmable interrupt controller (MPIC) system. The MPIC 100 consists of three major units: I/O-MPIC unit 102, MPIC-bus 103, and multiple local-MPIC units of which one is labelled 104. The I/O-MPIC 102 accepts interrupt lines 107 from its associated I/O subsystem 101 (typically a collection of peripherals), each line corresponding to a unique IRQ. The I/O-MPIC output is coupled to MPIC bus 103 which broadcasts to all local-MPIC units 104 appropriately formatted IRQ messages containing all necessary identifying and priority information. Each local MPIC unit 104 examines the message and decides whether to accept it. If tentatively accepted by more than one local MPIC unit 104, an arbitration procedure is invoked between competing units. The local MPIC unit 104 with the lowest priority wins the arbitration and accepts the IRQ and timely dispenses it to its associated processor 105.

System bus 30 is the common means for communicating between processors, memory, and other peripheral units of the multi-processor system. Each processor and peripheral is interlaced with system bus 30 by means of a memory bus controller (MBC) 31. In prior art systems, system bus 30 cams interrupt request traffic, interrupt servicing traffic; and all other inter-unit system traffic. The present invention relegates interrupt request traffic to the MPIC bus 103, thereby increasing the overall system efficiency.

B. Interrupt Control

The interrupt control functions of both I/O and local MPIC units are collectively responsible for delivering interrupts from interrupt sources to an interrupt servicing processor in a multi-processor system.

Each interrupt has an identity, the interrupt vector, that uniquely distinguishes the interrupt from other interrupts in the system. When a processor accepts an interrupt (IRQ), it uses the vector to locate the entry point of the appropriate software interrupt handler in its interrupt table. The preferred embodiment supports 256 (8-bit) distinct vectors in the range of 0 to 255.

Each interrupt has an interrupt priority represented by the five most significant bits of the 8-bit interrupt vector, i.e., 16 priority levels with 0 being lowest and 15 being the highest priority. This implies that 16 different vectors may share a single interrupt priority level.

Interrupts are generated by a number of different sources, which may include:

1) external I/O devices connected to the I/O-MPIC unit manifested by either edges (level transitions) or levels on interrupt input pins and may be redirected to any processor;

2) locally connected device interrupts, always directed to the local processor only, manifested as an edge or level signal;

3) MPIC timer interrupts, generated within the local-MPIC unit by any of the three programmable timers;

4) inter-processor interrupts addressed to any individual processor of groups of processors in support of software self interrupts, pre-emptive scheduling, cache memory table look-aside buffer (TLB) flushing, and interrupt forwarding; and

5) bus parity error interrupt generated by any local MPIC unit that detects a parity error on the data bus causing its host to be interrupted.

The destination of an interrupt can be zero, one, or a group of processors in the system. A different destination can be specified for each interrupt. The sender specifies destination of an interrupt in one of two destination modes: physical mode and logical mode.

In physical mode the destination processor is specified by a unique 8-bit MPIC-ID. Only a single destination or a broadcast to all (MPIC-ID of all ones) can be specified in physical destination mode.

Each MPIC unit has a register that contains the unit's 8-bit MPIC-ID. The MPIC-ID serves as a physical name of the MPIC unit. It can be used in specifying destination information and is also used for accessing the MPIC-bus. The mechanism by which an MPIC establishes its MPIC-ID is implementation dependent. Some implementations may latch in the MPIC-ID on some of their pins from slot number at reset time. The MPIC-ID is read-write by software.

The MPIC-ID serves as the physical "name" of the MPIC unit used for addressing the MPIC in physical destination mode and for MPIC-bus usage.

In logical mode, destinations are specified using a 32-bit destination field. All local MPIC units contain a 32-bit Logical Destination register 223 against which the destination field of the interrupt is matched to determine if the receiver is being targeted by the interrupt. An additional 32-bit Destination Format register 221 in each local-MPIC unit defines exactly how the destination field is to be compared against the destination format register. In other words, the Destination Format register 221 defines the interpretation of the logical destination information.

The Destination Format register 221 partitions the 32-bit destination information into two fields:

(1) an encoded tie Id that can be used to represent some scalar ID. Matching on the encoded field requires an exact match on the value of this field. To support broadcast to all in logical mode, an encoded field value of all ones is treated special in that it matches any encoded value.

(2) a decoded field (or bit array) that can be used to represent a sot of elements. Matching on the decoded field requires that at least one of the corresponding pair of bits in the decoded fields are both ones.

The Destination Format register 221 is controlled by software and determines which bits in the destination information are part of the encoded field and which bits are part of the decoded field. To have a match on the destination, both fields must match.

The logical-level interpretation of what each field really represents is totally defined by the operating system. Note that these fields need not use consecutive bits and that the length of either field can be zero. A zero-length field always matches. Since destination interpretation is done locally by each local-MPIC unit, the Destination Format registers of all local MPIC units in a system must be set up identically.

Three example usage models using different interpretations are described next to further illustrate the destination specification mechanism. These are probably the most common models used in practice.

EXAMPLE 1

Single-Level Model

In this model, all 32 bits of destination information are interpreted as decoded field. Each bit position corresponds to an individual Local MPIC unit. Bit position could correspond to physical MPIC-ID, but this need not be the case. This scheme allows the specification of arbitrary groups of MPIC units simply by setting the member's bits to one, but allows a maximum of 32 processors (or local-MPIC units) per system. In this scheme, a MPIC unit is addressed if its bit is set in the destination array. Broadcast to all is achieved by setting all 32 destination bits to one. This selects all MPIC units in the system.

EXAMPLE 2

Hierarchical Model

This model uses encoded and decoded fields of non-zero lengths. The encoded field represents a static cluster of local MPIC units, while a bit position in the decoded field identifies an individual local MPIC unit within the cluster. Arbitrary sets of processors within a cluster can be specified by naming the cluster and setting the bits in the decoded field for the selected members in the cluster. This supports systems with more than 32 processors, and matches a DASH-style cluster architecture. Broadcast to all is achieved by setting all 32 destination bits to one. This guarantees a match on all clusters, and will select all MPICs in each cluster.

EXAMPLE 3

Bimodal Model

Each value of the encoded field is the ID of an individual local-MPIC. This ID could be identical to the MPIC's physical MPIC-ID, but this need not be the case. Each bit in the decoded field represents a predefined group. This scheme allows addressing a single MPIC unit by using its ID in the encoded field (and selecting no groups), or to address a group (or union of groups) of MPICs by setting the encoded field to all ones and selecting the groups in the decoded field. Each MPIC unit could be a member of multiple groups. Supporting broadcast to all in the bimodal model requires that software define a group that contains all local-MPICs in the system, Broadcast is then achieved by setting all 32 destination bits to one. This matches all individual IDs and also matches on the group that contains 811 local units.

Each processor has a processor priority that indicates the relative importance of the task or code that the processor is currently executing. This code my be part of a process or thread, or it may be an interrupt handler. The priority is dynamically raised or lowered with changing tasks thus masking out lower priority interrupts. Upon servicing of an IRQ, the processor returns to a previously interrupted active.

A processor is lowest priority within a given group of processors if its processor priority is the lowest of all processors in the group. Because one or more processors may be simultaneously lowest priority within a given group, availability is subject to the process of arbitration.

A processor is the focus of an interrupt if it is currently servicing that interrupt, or if it currently has a request pending for that interrupt.

An important feature of the current invention is the guarantee of exactly-once delivery semantics of interrupts to the specified destination which implies the following attributes of the interrupt system:

1) interrupt injection is never rejected;

2) interrupts (IRQs) are never lost;

3) in the case of edge triggered interrupts, the same IRQ occurrence is never delivered more than once, i.e., by delivering an interrupt first to its focus processor (if it currently has one), multiple occurrences of the same interrupt while the first is pending (servicing not complete) are all recorded as pending in the local-MPIC interrupt request register's (IRR's) pending bit corresponding to that particular interrupt request;

4) for level activated interrupts, the state of the I/O-MPIC's interrupt pin is re-created at the destination local-MPIC's IRR pending bit whenever its state differs from the state of the I/O-MPIC interrupt input pin, the destination local-MPIC only initiating the same IRQ upon execution of an end-of-interrupt (EOI) signal, unless the processor explicitly raises its task priority.

The preferred embodiment supports two modes for the redirection of these incoming IRQ and for the selection of the destination processor: fixed static mode and dynamic lowest-priority mode. These and other possible operating system supported modes are supported by the following information:

1) MPIC-ID's, known by each MPIC unit,

2) Destination Address field from the I/O-MPIC's re-direction table,

3) the MPIC unit address; each MPIC unit knows its own address,

4) whether an MPIC unit is currently the focus for the interrupt, and

5) priority of all processors.

The fixed mode is the simplest method. The interrupt is unconditionally broadcast by the I/O MPIC to all MPIC's encoded in the destination address field for the particular IRQ, typically a single local MPIC is designated. Priority information is ignored. If the destination processor is not available, the interrupt is held pending at the destination processor's local-MPIC until the processor priority is low enough to have the local-MPIC dispense the interrupt to the processor. Fixed re-direction results in:

1) static distribution across all processors; and

2) assignment of a specific local-MPIC to a given interrupt.

Fixed redirection allows existing single threaded device drivers to function in a multi-processor environment provided that software binds the driver code to run on one processor and the MPIC unit is programmed for fixed delivery mode so that the device's interrupt is directed to the same processor on which the driver runs,

The lowest-priority re-direction mode causes the lowest priority available processor in a group specified by the re-direct address field to service the interrupt. Because each of these lowest-priority processors local-MPIC knows their associated processor's priority, an arbitration protocol is exercised on the MPIC-bus to determine the lowest priority.

If more than one processor is operating at the lowest-priority, then one of them my be picked at random. An additional processor selection algorithm is applied to the remaining candidate lowest priority processors for the random selection of a processor with the object of uniformly spreading the interrupt servicing task amongst the lowest priority processors.

C. Structural Description

The I/O-MPIC unit 102 of FIG. 2 is further detailed in FIG. 3. The interrupt input lines 107 provide the means for the I/O devices to inject their interrupts. An edge filter 108 is used to provide dean level transitions at the input pins. The re-direction table 104 has a dedicated 64-bit entry for each interrupt input pin (line) 107. Unlike the prior art IRQ pins of the 82C59A/82380 PIC previously discussed, the notion of interrupt priority is completely unrelated to the position of the physical interrupt input pin on the I/O MPIC unit of the current invention. The priority of each input pin 107 is software programmable by assigning an 8-bit vector in the corresponding entry of the Re-direction Table 104.

FIG. 4 shows the format of each Re-direction Table 64-bit entry. The description of each entry is as follows:

Vector (0:7):	An 8-bit field containing the interrupt vector.
Delivery Mode (8:10):	A 3-bit field that specifies how the local MPIC's listed in the destination field should act upon receipt of this signal, has following meaning:
000-	Fixed - deliver to all processors listed on destination.
001-	Lowest Priority - deliver to lowest priority processor among all processors listed in destination.
011-	Remote Read - request contents of an MPIC unit register, whose address is in the Vector field, to be stored in the Remote Register for access by the local processor, edge trigger mode.
100-	NMI - deliver to the non-maskable interrupt (NMI) pin of all listed processors, ignoring vector information, treated as edge sensitive signal.
101-	Reset - deliver to all processors listed by asserting/de-asserting the processors' reset pin, setting all addressed local
110-	Debug - deliver to all listed processors by asserting/de-asserting the local MPIC's debug pin; treated as a level sensitive signal.
111-	Ext INT - deliver to the INT pin of all listed processors as an interrupt originating in an externally connected 8259A compatible interrupt controller, treated as a level sensitive signal.
(Note that Delivering Modes of Reset, Debug and ExtINT are not I/O device interrupt related. Reset and Debug are interprocessor interrupts while the ExtINT mode is included to provide compatibility with the existing 8259A PIC de facto standard.)	
Destination Mode (11):	Interprets Destination field: 0 - Physical Mode - use MPIC-ID in bits 56:63. 1 - Logical Mode - 32 bit field is the logical destination, operating system defined.
Delivery Status (12):	A 2-bit software read only field containing current delivery status of the interrupt:
0-	Idle - no current activity,
1-	Send Pending - interrupt injected to local-MPIC, held up by other injected interrupts, This bit is software read-only, i.e., 32-bit software writes to the re-direction table 109 do not affect this bit.
Remote IRR (14):	Mirrors the Interrupt Request Register (IRR) bit of the destination local-MPIC for level sensitive interrupts only, and when status of bit disagrees with the state of the corresponding interrupt input line 107, an I/O-MPIC message is sent to make the destination's IRR bit reflect the new state causing the remote (local-MPIC) IRR bit to track. This bit is software read-only.
Trigger Mode (15):	Indicates the format of interrupt signal: 0 - edge sensitive 1 - level sensitive
Mask (16):	Indicates mask status: 0 - non masked interrupt (NMI) 1 - masked interrupt capable of being blocked by higher priority tasks.
Destination (32:63):	32-bit field representing interrupt destination defined by the operating system. The lower part of FIG. 4 depicts the two possible formats previously discussed: a 32-bit physical (decoded) format using one bit per destination processor and an 8/24 bit logical format with 8 coded bits and 24 decoded bits defining a 256 × 24 two dimensional destination space.

The 64-bit wide re-direct table 109 is read/write accessible through the 32-bit address and 32 data lines, DATA/ADDR 106, of a host processor, except as noted above for the Delivery Status and Remote IRR bits which are hardware write and software reed only.

The re-direction table entries are formatted and broadcast to all local-MPIC units 104 by MPIC-bus send/receive unit 110. The MPIC-bus 107 protocol specifies a 5-wire synchronous bus, 4 wires for data and one wire for its clock. Specific details of message formats will be covered under the section on MPIC-Bus Protocol. Acceptance results in the reset of the Delivery Status to Idle.

The local-MPIC unit 104 is responsible for interrupt acceptance, dispensing of interrupts to the processor and sending inter-processor interrupts.

Depending on the interrupt delivery mode specified in the interrupt's re-directional table entry, zero, one or more MPIC units may accept an interrupt. A local-MPIC accepts an interrupt only if it can deliver the interrupt to its associated processor. Accepting an interrupt is purely an I/O-MPIC 102 and local-MPIC 104 matter while dispensing an interrupt to a processor only involves a local-MPIC 104 and its local processor 105.

The Re-direction Table 109 of the I/O-MPIC unit 102 serves to steer interrupts that originate in the I/O sub-system 101 and may have to be directed to any processor by broadcasting the Re-direction Table entry corresponding to a given interrupt over the MPIC-Bus 103.

FIG. 5 details the structural elements of the local-MPIC unit 104. The Local Vector Table 210 is similar in function to the I/O-MPIC Re-direction Table 109 except that it is restricted to interrupts relating to the associated local processor only. Local Vector Table 210 contains six 32-bit entries. Entries 201 through 202 correspond to timers 0 through 2; entries 203 and 204 correspond to local interrupt input pins; and entry 205 controls interrupt generation for data parity errors. The higher order bits in timer entries 200 through 202 contain timer-specific fields not present in the other entries (as detailed under the later discussion on timers).

Although FIGS. 2 and 5 show local MPIC unit 104 as a separate entity, it may be incorporated in whole or in pert into the associated processor or processor chip 105. This may be done in order to improve the efficiency of communications between the local MPIC unit 104 and the associated processor 105. For example, the incorporation of the local MPIC unit's local interrupt vector table comprising units 203 and 204 could provide a more direct path to cache memory and thus speed up the flushing of a cache memory translation look-up buffer.

FIG. 6 defines the various fields associated with local vector table entries 200 through 205.

Vector (0:7):	An 8-bit field containing the interrupt Vector.
Delivery Mode (DELV)(8:10):	A 3-bit field having the same meaning as in re-direction table 109 except lowest priority (001) is synonymous with fixed (000).
Remote IRR (R) (14):	This bit mirrors the interrupt's IRR bit of this local-MPIC unit. It is used solely for level triggered local interrupts, is undefined for edge triggered interrupts, and is software read-only.
Trigger Mode (TM) (15):	0 indicates edge sensitive trigger. 1 indicates level sensitive interrupt. Local interrupt pins (203, 204) may be programmed as other edge or level triggered while Timer (200:202) and Parity (205) are always edge sensitive.
Mask (MS)(16):	0 enables interrupt, 1 masks interrupt
Mode (M)(17):	Selects mode of timer; 0 is one shot, 1 is periodic
Base (18:19):	Selects one of three time base for counter.

(Mode and Base parameters will be discussed further in the section on Timer Architecture.)

A processor generates inter-processor interrupts by writing to the 64-bit interrupt Command Register 220, whose layout is similar to that of the I/O-MPIC Re-direction Table 109. The programmable format, very similar to an entry in the Re-direction Table 109 is shown in FIG. 7, allows every processor to generate any interrupt thereby allowing a processor to forward an interrupt originally accepted by it to other processors. This feature is also useful for debugging. The Interrupt Command Register 220 is software read-write.

Vector (0:7):	Identifies interrupt being sent.
Delivery Mode (8:10):	Same interpretation as for Re-direction Table 109.
Destination Mode (11):	Same interpretation as for Re-direction Table 109.
Delivery Status (12):	Same interpretation as to Re-direction Table 109. Local processor sets status, local-MPIC up-dates. Software may read this field to find out if the interrupt has been sent and, if so, Interrupt Command Register 220 is ready for accepting a new interrupt. If register 220 is over-written before the delivery status is Idle (0), then the status of that interrupt is undefined (may or may not have been accepted).
Level De-assert (14):	A bit used in conjunction with Trigger Mode (15) to simulate assertion/de-assertion of level sensitive interrupts (0-de-assert, 1-assert). For example, with Delivery Mode at Reset, Trigger Mode at Level and Level De-assert at 1, results in a de-assert of Reset to the processor of the addressed MPIC(s). This condition will also cause all MPICs to reset their Arbitration-ID (used for tie breaking in lowest priority arbitration) to the MPIC-ID.
Trigger Mode (15):	Same as for Redirection Table 109.
Remote Read Status (16:17)	Indicates status of data contained in the Remote Read register 224: 00 - Invalid - content of Remote Read register 224 invalid, remote MPIC unit unable to deliver. 01 - In Progress - Remote Read in progress, awaiting data. 10 - Valid - Remote Read complete, valid data.
Destination Shorthand (18:19):	A 3-bit field used to specify a destination without the need to provide the 32-bit destination field. This reduces software overhead by not requiring a second 32-bit write operation corresponding to bit field 32:63 for the follow common cases: a) software self interrupt, b) interrupt to a single fixed destination, c) interrupt to all processors that can be named in the Destination field (32:63), including the sending processor. The 2-bit code is interpreted as follows: 00 - No shorthand, use Destination field (32:63). 01 - Self, current local-MPIC is the only destination (used for software interrupts). 10 - All including self. 11 - All excluding self, used during reset and debug.
Destination (32:63):	Operating system defined, same as for Redirection Table 109. Used only when Destination Shorthand is set to Dest Field (000).

The I/O-MPIC unit 102 and all local MPIC units 104 receive messages via the MPIC-Bus 103. The MPIC unit's firm check to see if it belongs to the destination in the message. For example, in the case of the 32-bit destination format previously cited, each MPIC unit with an ID value in MPIC-ID Register 222 less than 32 uses it MPIC-ID to index into the 32-bit destination array. If it finds its bit set, then the MPIC unit is addressed by this message. In the case of the 8×24 format, each MPIC unit checks if its MPIC-ID is equal to the MPIC-ID in the 32-bit destination field, or if it is a member of the group list, as shown in FIG. 7, by bit-wise ANDing its 24-bit group list register (32:55) with the group list in the message and ORing all resulting bits together. If the MPIC-ID in the message has a value of 255, then the MPIC unit is addressed by the message as well.

MPIC-bus send/receive and arbitration unit 226. FIG. 5, directs the destination and mode information on output 267 to acceptance logic unit 248 which performs the logic operations in conjunction with the contents of MPIC-ID Register 222. If the message is accepted, the vector information available on output 266 of unit 226, is decoded and together with the mode information is passed on through to the 3×256 bit vector array 230 by vector decode unit 228. The 8-bit interrupt Vector, when decoded by Vector Decode 228 determines which bit position out of 256 possible is set indicating the interrupt priority. When an interrupt is being serviced, all equal or lower priority interrupts are automatically masked by the prioritizer unit 240.

3×256 bit Vector Array 230 consists of 256 bit vectors used for storing interrupt related information. Each register is software road-only and hardware road/write. The registers are defined as follows:

ISR,	In Service Register 231, shows interrupts that are currently in service for which no end of interrupt (EOI) has

-continued

	been sent by the processor;
IRR,	Interrupt Request Register 232, contains interrupts accepted by the local MPIC unit but not dispensed to the processor,
TMR,	Trigger Mode Register 234, indicates whether the interrupt is a level or edge sensitive type as transmitted by the sending MPIC-I/O units trigger mode bit in the redirection table entry.

If an interrupt goes in service and the TMR bit is 0, indicating edge type, then the corresponding IRR bit is cleared and the corresponding ISR bit is set. If the TMR bit is 1, indicating level type, then the IRR bit is not cleared when the interrupt goes in service (ISR bit set). Instead, the IRR bit mirrors the state of the interrupt's input pin. As previously discussed, when the level triggered interrupt is de-asserted, the source I/O-MPIC detects the discrepancy and sends a message to the destination local-MPIC unit to clear its IRR bit.

FIG. 8 shows, by way of an example, how the remote IRR and the IRR bit at the destination local-MPIC unit track the state of the interrupt input (INTIN). It else illustrates how an EOI is followed immediately by the re-assertion of the interrupt as long as the INTIN is still asserted by some device. In this example, it is assumed that two devices, A and B, sham a level triggered interrupt input to the I/O-MPIC. Device A raises a level interrupt as indicated on line (a), followed by a device B interrupt as shown on line (b). The resulting INTIN signal is the ORed combination of lines (a) and (b) as shown on line (c). The MPIC-bus send/receive unit 110 of FIG. 3 EXCLUSIVE-ORs (XORs) INTIN with the remote IRR bit, bit 14 of the Re-direction Table 104, shown on line (e) to yield the "level assert" and "level de-assert" shown on line (d). The local-MPIC IRR bit tracks the state of line (e) as shown on line (f). Line (g) demonstrates how an EOI is followed immediately by re-asserting the interrupt as long as the INTIN signal is still asserted by one of the devices.

FIG. 9 is a flow chart depicting the interrupt acceptance process of a local-MPIC unit. Upon receipt of a message, a local-MPIC unit is the current focus, i.e., the pertinent IRR or ISR bit is pending, it accepts the interrupt independent of priority and signals the other local-MPICs to abort the priority arbitration. This avoids multiple delivery of the same interrupt occurrence to different processors, consistent with prior art interrupt delivery semantics in uni-processor systems. If a local-MPIC unit is not, currently, the focus it listens for the acceptance by another local-MPIC. If more than one MPIC is available, arbitration, as described under the section entitled MPIC Bus Protocol, is invoked to determine the winner (lowest priority) unit.

If a message is sent as NMI, Debug or Reset, then all units listed in the destination unconditionally assert/de-assert their processor NMI output pin 263, Debug pin 264, or Reset pin 265 of FIG. 5. ISR 231 and IRR 232 are by-passed and vector information is undefined.

The Task Priority Register (TPR) 242 of FIG. 5 stores the current priority of its processor's task which is dynamically subject to change because of explicit software actions, such as when tasks are switched, and upon entering or returning from an interrupt handler. TPR 242 is a 32-bit register supporting up to 256 priority levels by means of an 8-bit field (0:7). The four msb (4:7) correspond to the 16 interrupt priorities while the four lsb (0:3) provide additional resolution. For example, a TPR value with zero in the five msb and non-zero in the three lsb may be used to describe a task scheduling class between 0 (Idle) and 1 for the purpose of assigning an interrupt. This is particularly useful when a number of processors are operating at the same lowest level of priority.

A processor's priority is derived from the TPR 242, ISR 231 and IRR 232. It is the maximum of its task priority, and the priority of the highest order ISR bit, and the priority of the highest IRR bit, all evaluated using the four most significant bits of their coded 8-bit representation. This value, used in determining availability of a local-MPIC to accept an interrupt and in determining the lowest priority local-MPIC unit, is computed on the fly as required.

Once a local-MPIC accepts an interrupt, it guarantees delivery of the interrupt to its local processor. Dispensing of a maskable interrupt is controlled by the INT/INTA protocol which begins with the local-MPIC unit asserting the INT pin 262 which is connected to the processor INT pin. If the processor has interrupts enabled, it will respond by issuing an INTA cycle on line 261 causing the local-MPIC to freeze its internal priority state and release the 8-bit Vector of the highest priority interrupt onto the processor data bus-106. The processor reads the Vector and uses it to find the interrupt handler's entry-point. The local-MPIC also sets the interrupt's ISR-bit. The corresponding IRR bit is cleared only if TMR 234 indicates an edge triggered interrupt as previously discussed.

If a level triggered interrupt is de-asserted just prior to its INTA cycle, all IRR bits may be dear end the Prioritizer 240 may not find a Vector to deliver to the processor on data bus 106. Instead, the Prioritizer 240 will return a Spurious Interrupt Vector (SIV) instead. The dispensing of the (SIV) does not effect ISR 231, so that the interrupt handler should return without issuing an EOI. The SIV is programmable via the SIV register within prioritizer 240.

It is possible that local MPIC units exist in the system that de not have a processor to which to dispense interrupts. The only danger this represents in the system is that if an interrupt is broadcast to all processors using lowest priority delivery mode when all processors are at the lowest priority, there is a chance that a local MPIC unit without the processor may accept the interrupt if this MPIC unit happens to have the lowest Arb ID at the time. To prevent this from happening, all local units initialize in the disabled state and must be explicitly enabled before they can start accepting MPIC messages from the MPIC-bus. A disabled local MPIC Unit only responds to messages with Delivery Mode set to "Reset". Reset/deassert messages should be sent in Physical Destination mode using the target's MPIC ID because the logical destination information in the local MPICs is undefined (all zeroes) when the local-MPIC comes out of Reset.

Before returning from an interrupt handler, software must issue an end-of-interrupt (EOI) command to its local-MPIC clearing the highest priority bit in ISR 231, by writing to EOI register 246, indicating the interrupt is no longer in service, and causing it to return to the next highest priority activity.

The MPIC system is initialized in the following manner:

a) Each MPIC unit has a reset input pin connected to a common reset line and activated by the system reset signal.

b) The 8-lsb of the data bus 106 are latched into MPIC-ID register 222;

c) Each local-MPIC asserts its processor Reset (RST) pin 265 and resets all internal MPIC registers to their initial state, i.e., Re-direction Table 109 and Local Vector Table 210 set so as to mask interrupt acceptance, otherwise setting register state to zero;

d) Each local-MPIC de-asserts it processor's reset pin to allow the processor to perform serf-test and execute initialization code;

e) The first processor to get on MPIC-bus 103 will force other processors into reset by sending them the inter-processor interrupt with all other processors being kept in reset until the active processor's operating system allows them to become active;

Delivery Mode	= Reset
Trigger Mode	= Level.
Level De-Assert	= 0
Destination Shorthand	= All Excl Self,

f) The only running processor performs most of the system initialization and configuration, eventually booting an operating system which sends out a De-assert/Reset signal to activate the other processors.

D. MPIC-Bus Protocol

The MPIC-bus 103 is a 5-wire synchronous bus connecting I/O-MPIC and local-MPIC units. Four of these wires are for data transmission and arbitration while one is a clock line.

Electrically, the bus is wire-OR connected providing both bus-use arbitration, and lowest priority arbitration. Because of the wire-OR connection, the bus is run at a low enough speed such that design-specific termination tuning is not required. Also, the bus speed must allow sufficient time in a single bus cycle to latch the bus and perform some simple logic operations on the itched information in order to determine if the next drive cycle must be inhibited. With a 10 MHz bus speed, an interrupt requiring no arbitration would be delivered in about 2.3 μs, and with priority arbitration, about 3.4 μs.

The MPIC units 102 and 104 have separate MPIC-bus input and output pins which may be directly connected in a non-isolated configuration as shown in FIG. 11. Three-state input buffers 301 and output buffers 302 may be used to provide a hierarchical connection to MPIC-buses that are required to support a large number of processors as shown in FIG. 12.

Arbitration for use of the MPIC-bus 103 and for determining the lowest priority MPIC unit depends on all MPIC message units operating synchronously. Distributed bus arbitration is used to deal with the case when multiple agents start transmitting simultaneously. Bus arbitration uses a small number of arbitration cycles on the MPIC-bus. During these cycles, arbitration "losers" progressively drop off the bus until only one "winner remains transmitting. Once the sending of a message (including bus arbitration) has started, any possible contender must suppress transmission until enough cycles have elapsed for the message to be fully sent. The number of bus cycles used depends on the type of message being sent.

A bus arbitration cycle starts with the agent driving its MPIC-ID on the MPIC-bus, higher order bits first. More specifically, the 8-bit MPIC-ID (10:17) is chopped into successive groups of 2 bits (17:16) (15:14) (13:12) (11:10). These tuples I(im):I(i), are then sequentially decoded to produce a four bit pattern (B0:B3) as shown in FIG. 13. Bits (B0:B3) are impressed on the four MPIC-bus lines, one bit per line. Because of the wired-OR connection to the MPIC-bus, each tuple of the ID only asserts to a single wire, making it possible for an agent to determine with certainty whether to drop-off ("lose") or to continue arbitrating the next cycle for the following two bits of the MPIC-ID, by simply checking whether the bus line agent that is driving the bus is also the highest order 1 on the bus. In this manner, each MPIC-Bus cycle arbitrates two bits.

Arbitration is also used to find the local MPIC unit with the lowest processor priority. Lowest-priority arbitration uses the value of the local MPIC unit's Task Priority Register appended with an 8-bit Arbitration ID (Arb ID) to break lies in case them are multiple MPICs executing at the lowest priority.

Using the constant 8-bit MPIC ID as the Arb ID has a tendency to skew symmetry since it would favor MPICs with low ID values. An MPIC's Arb ID is therefore not the MPIC ID itself but is derived from it. N reset, an MPIC's Arb ID is equal to its MPIC ID. Each time a message is broadcast over the MPIC-Bus, 811 MPICs increment their Arb ID by one, which gives them a different Arb ID value for the next arbitration. The Arb ID is then endian-reversed (LSB becomes MSB, etc.) to ensure more random selection of which MPIC gets to have the lowest Arb ID next time around. The reversed Arb ID is then decoded to generate arbitration signals on the MPIC bus as described above.

After bus arbitration the winner will drive its actual message on the bus. 4 bits per clock in nibble-serial fashion. MPIC messages come in two lengths: short at 21 cycles and long at 30 cycles. The interpretation of the first 19 cycles is the same for all message lengths. The long message type appends cycles for priority arbitration to the first 19 cycles. The medium message type only occurs if a complete arbitration is not needed as in case when the winner is known prior to arbitration.

The short message format is shown in FIG. 14 where the first column represents the message cycle index (1:19) while the next four columns represent the 4 data lines of the MPIC-bus.

The first four rows (1:4) represent the MPIC-bus arbitration cycles where any row has four entries representing the decoded tuple of the MPIC-ID as previously discussed, i.e., i76 . . . 176 represents tuple (7:6), i54 . . . i54, tuple (5:4), etc.

Cycle 5 is the extended delivery mode of the message and is interpreted in accordance with FIG. 15. The bit designated DM is the Destination Mode bit which is Ø for Physical Mode and 1 for Logical Mode. Bits M0, M1, M2 are previously assigned in the Redirection Table of FIG. 4.

Cycle 6 contains the control bits as defined in FIG. 16. The extended delivery mode and control bits, cycles 5 and 6, together determine the length of message needed and the interpretation of the remaining fields of the message as shown in FIG. 17.

Cycles 7 and 8 make-up the 8-bit interrupt vector.

Cycles 9 through 16 are the 32-bit destination field.

Cycle 17 is a checksum over the data in cycles 5 through 16. The checksum protects the data in these cycles against transmission errors. The ending MPIC unit provides this checksum.

Cycle 18 is a postamble cycle driven as 1111 by the sending MPIC allowing all MPICs to perform various internal computations based on the information contained in the received message. One of the computations takes the computed checksum of the data received in cycles 5 through 16 and compares it against the value in cycle 17. If any MPIC unit computes a different checksum than the one passed in cycle 17, then that MPIC will signal an error on the MPIC-bus in cycle 19 by driving it as 1111. If this happens, all MPIC units will assume the message was never sent and the sender must try sending the message again, which includes re-arbitrating for the MPIC-bus. In lowest priority delivery when the interrupt has a focus processor, the focus processor will signal this by driving 1110 during cycle 19. This tells all the other MPIC units that the interrupt has been accepted, the arbitration is preempted, and short message format is used. All (non focus) MPIC units will drive 1000 in cycle 19. Under lowest priority delivery mode, 1000 implies that the interrupt currently has no focus processor and that priority arbitration is required to complete the delivery. In that case, long message format is used. If cycle 19 is 1000 for non Lowest Priority mode, then the message has been accepted and is considered sent.

When an MPIC unit detects and reports an error during the error. cycle, that MPIC unit will simply listen to the bus until it encounters two consecutive idle (0000) Cycles. These two idle Cycles indicate that the message has passed and a new message may be started by anyone. This allows an MPIC that got itself out of Cycle on the MPIC-bus get back in sync with the other MPIC units.

Cycles 1 through 19 of the long message format am identical to Cycles 1 through 19 of the short message format.

As mentioned, long message format is used in two cases:

(1) Lowest priority delivery when the interrupt does not have a focus. Cycles 20 through 27 are eight arbitration Cycles where the destination MPIC units determine the one MPIC unit with lowest processor priority/Arb ID value.

(2) Remote Read messages. Cycles 20 through 27 are the 32 bit content of the remotely read register. This information is driven on the bus by the remote MPIC unit.

Cycle 28 is an Accept Cycle. In lowest priority delivery, all MPIC units that did not win the arbitration (including those that did not participate in the arbitration) drive Cycle 28 with 1100 (no accept), while the winning MPIC unit drives 1111. If cycle 28 reads 1111, then all MPIC units know that the interrupt has been accepted and the message is considered delivered. If cycle 28 reads 1100 (or anything but 1111 for that matter), then all MPIC units assume the message was unaccepted or an error occurred during arbitration. The message is considered undelivered, and the sending MPIC unit will try delivering the message again.

For Remote Read messages, cycle 28 is driven as 1100 by all MPICs except the responding remote MPIC unit, that drives the bus with 1111, if it was able to successfully supply the requested data in cycles 20 through 27. If Cycle 28 reads 1111 the data in Cycles 20 through 27 is considered valid; otherwise, the data is considered invalid. The source MPIC unit that issued the Remote Read uses Cycle 28 to determine the state of the Remote Read Status field in the Interrupt Command Register (valid or invalid). In any case, a Remote Read request is always successful (although the data may be valid or invalid) in that a Remote Read is never retried. The mason for this is that Remote Read is a debug femurs, and a "bung" remote MPIC that is unable to respond should not cause the debugging procedure to hang.

Cycles 29 end 30 are two idle cycles. The MPIC bus is available for ending the next message at cycle 31. The two idle cycles at the end of both short and long messages, together with non zero (i.e., non idle) encodings for certain other bus cycles allow an MPIC bus agent that happens to be out of phase by one cycle to sync back up in one message simply by waiting for two consecutive idle cycles after reporting its checksum error. This makes use of the fact that valid arbitration cycles am never 0000.

E. Timers

The Local Vector Table 210 of the local-MPIC unit 104 contains three independently operated 32-bit wide programmable timers 200, 201 and 202. Each timer can select its clock base from one of three clock inputs. Each time may operate in either a one-shot mode or a periodic mode and each can be configured to interrupt the local processor with an arbitrary programmable vector.

The local-MPIC unit 104 has two independent clock input pins: CLOCK pin provides the MPIC's internal clock, TMBASE provides for an external clock. The frequency of TMBASE is fixed by the MPIC architecture at 28.636 MHz. In addition, the local-MPIC contains a divider that can be configured to divide either clock signal by 2, 4, 8 or 16, as shown in FIG. 19. Base 0 is always equal to CLOCK; Base 1 always equals TMBASE; and Base 2 may either equal CLOCK or TMBASE divided by 2, 4, 8 or 16. Divider (Base 2) Configuration Register is shown in FIG. 20.

Software starts a timer by programming its 32-bit Initial Count Register. The timer copies this value into the Current Count Register and starts counting down at the rate of one count for each time base pulse (Base 0, 1 or 2). Each timer may operate One-shot or Periodic. If One-shot, the timer counts down once and remains at zero until reprogrammed. In Periodic Mode, the timer automatically reloads the contents of the Initial Count Register into the Current Count Register.

The three timers are configured by means of their Local Vector Table entries as shown in FIG. 21. The Vector field (0:7) is as previously described. The Mask i, bit (16) serves to mask (1) or not mask mask (0), the i^{th} timer generated interrupt when count reaches 0. Base i field (18:19) is the base input used by the i^{th} timer: 00-Base 0, 01-Base 1, and 10-Base 2. Mode i bit (17) indicates the mode of the i^{th} timer: 0-one-shot, 1-periodic.

F. Processor Private Storage

Each local-MPIC unit provides Processor Private Storage 250, as shown in FIG. 5, with four 32-bit registers accessible only by the local processor. Since each processor addresses its registers in the came way (via the same address), the registers provides a convenient and processor architecture independent way of providing "processor-own" data. The contents of these registers is not interpreted by the MPIC in any way. These registers are located in the same physical address page as the other local MPIC registers, access to these registers may therefore be restricted to supervisor only. The operating system running on the processor is free to use these registers as it pleases.

We claim the following:

1. A multi-processor (MP) system comprising:

a plurality of processors;

a system bus, the plurality of processors communicating on the system bus;

an interrupt bus having wired-OR data lines;

an input/output (I/O) interrupt controller that broadcasts an interrupt request on the interrupt bus;

a plurality of local interrupt controllers coupled to the interrupt bus, each of the local interrupt controllers being associated with a corresponding processor of the MP system, each local interrupt controller being operable to perform an arbitration priority sequence on the wired-OR data lines to arbitrate for acceptance of the interrupt request, the arbitration priority sequence including driving a current priority value associated with the corresponding processor on the wired-OR data lines of the interrupt bus.

2. The MP system of claim 1, wherein the current priority value associated with the corresponding processor comprises either a current task priority, a pending highest-priority interrupt request, or a serviced highest-priority interrupt request.

3. The MP system of claim 1, wherein the current priority value associated with the corresponding processor comprises a maximum value of either a current task priority, a pending highest-priority interrupt request, or a serviced highest-priority interrupt request.

4. The MP system of claim 1, wherein the current priority value associated with the corresponding processor includes an appended arbitration ID of the local interrupt controller.

5. The MP system of claim 1, wherein the interrupt request includes an indication of a focus processor, the indication preempting the arbitration priority sequence.

6. The MP system of claim 1, wherein each of the local interrupt controllers includes interrupt bus circuitry that drives the current priority value onto the interrupt bus, higher-order bits first.

7. A multi-processor (MP) system operating in accordance with a protocol for handling interrupts, the MP system comprising:

an interrupt bus;

a system bus;

a plurality of processors that communicate across the system bus;

a plurality of interrupt controller units (ICUs), each of the ICUs being coupled to the interrupt bus and a corresponding processor, each ICU including a first register that stores a task priority value of the corresponding processor, a second register that stores an identification (ID) of the ICU, and logic which receives interrupt messages on the interrupt bus and dispenses interrupts to the corresponding processor in accordance with the protocol, the protocol including first and second modes of operation,

in the first mode of operation an interrupt request is accepted by the ICU when a destination field of the interrupt request matches the ID of the ICU; and

in the second mode of operation the interrupt request is accepted by the ICU when a current priority value, which includes the task priority value, is a lowest current priority value from among the plurality of processors as determined by a bus arbitration sequence having a number of arbitration cycles.

8. The MP system of claim 7, wherein the first mode is a fixed, static mode of operation, and the second mode is a dynamic, lowest-priority mode of operation.

9. The MP system of claim 8, wherein the current priority value comprises the task priority value of the corresponding processor appended with an arbitration ID of the ICU.

10. The MP system of claim 9, wherein the arbitration ID is equal to the ID of the ICU at a reset state of the MP system, the arbitration ID of each ICU being incremented and endian-reversed each time a message is broadcast on the interrupt bus.

11. The MP system of claim 8, wherein the interrupt bus includes wired-OR data lines, during the bus arbitration sequence the logic of the ICU driving the current priority value on the wired-OR lines.

12. The MP system of claim 11, wherein during the bus arbitration sequence the logic of each ICU further checks the wired-OR lines to determine whether another processor has the lowest current priority value; if so, the logic drops-off the bus arbitration sequence; otherwise, the logic continues to arbitrate in a next arbitration cycle, or until the ICU determines that the current priority value is the lowest current priority value.

* * * * *